НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.052.42

До захисту допущено:
Завідувач кафедри
Сергій СТІРЕНКО
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Комп'ютерні системи та мережі»

зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Метод стиснення зображення на основі нейронної мережі**»**

Виконав (-ла):
студент (-ка) VI курсу, групи ІВ-91мн
Василенко Дмитро Євгенійович _____

Керівник:
професор кафедри ОТ, д.ф.-м.н.
Гордієнко Юрій Григорович _____

Консультант з нормоконтролю:
професор кафедри ОТ, д.т.н.
Кулаков Юрій Олексійович _____

Рецензент:
доцент кафедри АУТС, к.т.н
Писаренко Андрій Володимирович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність *123. Комп'ютерна інженерія*
(код і назва)

Спеціалізація *123. Комп'ютерні системи та мережі*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
«_____» _____ 2021 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Василенко Д.Є
(прізвище, ім'я, по батькові)

1. Тема дисертації Метод стиснення зображення на основі нейронної мережі

Науковий керівник дисертації професор, д.ф.-м.н. Гордієнко Ю.Г.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «12» 03 2021 р. № 809-с

2. Строк подання студентом дисертації _____

3. Об'єкт дослідження: метод стиснення зображення на основі нейронної мережі

4. Предмет дослідження: ефективність роботи методу компресії на зображеннях

5. Перелік завдань, які потрібно розробити: розглянути стандартні методи стиснення зображення та існуючі аналоги нейронних мереж розробити систему компресії та модель нейронної мережі; підготувати датасет;

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Кулаков Ю. О.		
1			
2			
3			
4			

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1	Затвердження теми роботи	17.09.2020	
2	Складання і узгодження технічного завдання	28.09.2020	
3	Написання вступної частини та огляд рішень	15.10.2020	
4	Розробка способу	22.12.2020	
5	Програмна реалізація	18.01.2021	
6	Оформлення пояснювальної записки	22.02.2021	
7	Попередній захист та проходження нормативного контролю	07.05.2021	
8	Подання МД рецензенту	11.05.2021	
9	Захист	17.05.2021	

Студент _____ Василенко Д. Є.
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ Гордієнко Ю. Г.
(підпис) (ініціали, прізвище)

РЕФЕРАТ
на магістерську дисертацію
виконану на тему:

Метод стиснення зображення на основі нейронної мережі

студентом: Василенко Дмитром Євгенійовичем

Робота складається із вступу та чотирьох розділів. Загальний обсяг роботи: 86 аркушів основного тексту, 42 ілюстрацій, 12 таблиця. При підготовці використовувалася література з 42 різних джерел.

Актуальність. З розвитком мережевих технологій кількість даних що використовуються у повсякденному житті кожним із нас зростає із неймовірною швидкістю, за нещодавніми дослідженнями за останні 20 років людство згенерувало більше інформації ніж за все своє існування, що ставить перед нами задачу ефективного способу збереження та передачі даних як одну із найпріорітетніших в даний момент.

Особливо важливим типом інформації є зображення і вміння його ефективно зберігати та передавати надає неймовірні можливості людству, проте останні роки ми користувалися лише класичними алгоритмами для стискання зображення. Отже використання нейронних мереж у даній сфері фактично може відкрити нові раніше не бачені горизонти що дозволяють нам перейти на нову ланку у розвитку технологій компресії.

Мета і завдання дослідження. Метою магістерської роботи є створення та розвиток способів покращення характеристик методу компресії зображення що базується на основі нейронної мережі.

Для досягнення поставленої магістерською роботою мети було поставлено й вирішено наступні завдання:

- дослідження поняття зображення та його структури й існуючих сучасних форматів різних типів;
- глибоко розглянуті існуючі методи компресії зображення із втратами та без втрат;

- розглянуто існуючі методи компресії зображення із використання нейронних мереж;
- проаналізовано та покращено існуючий метод глибокої компресії зображення що базується на основі квантування даних зображення у каналах;
- проілюстровано роботу моделі та проаналізовано результати створеного покращеного варіанту нейронної мережі відносно оригінального дослідження;

Об'єкт дослідження. Метод стиснення зображення на основі нейронної мережі

Предмет дослідження. Ефективність роботи методу компресії зображення та шляхи її покращення

Методи досліджень. Для досягнення поставлених в магістерській роботі задач, було використано методи машинного навчання та тренування нейронних мереж. Наукова новизна проведеного дослідження забезпечена наступними пунктами:

- було запропоновано новий блок нейронної мережі що є складовою кодувальника у підході до глибокої компресії зображення.
- розроблено та доповнено програмний продукт нейронної мережі для забезпечення поставленої задачі.
- тренування проведено на хмарному провайдері Microsoft Azure.

Особистий внесок здобувача. Магістерське дослідження є самостійно виконаною роботою, в якій відображено особистий авторський підхід та особисто отримані теоретичні та прикладні результати, що відносяться до вирішення задачі глибокої компресії зображення з допомогою нейронної мережі. Формулювання мети та завдань дослідження проводилось спільно з науковим керівником.

Практична цінність. Отримані результати можуть бути вільно використані у мабутніх дослідженнях за напрямками:

- машинне навчання
- глибока компресія зображення
- компресія зображення нейронними мережами

Публікації. Результати дослідження, яке було виконано в рамках дисертації, були представлені на наступних конференціях, та опубліковані в збірниках доповідей цих конференцій:

1. D. Vasylenko, S. Stirenko, Y. Gordienko, Deep Image Compression System for the Better Rate-Distortion Performance, The International Conference on Security, Fault Tolerance, Intelligence (ICSFTI2021 Online), Kyiv, Ukraine.
2. D. Vasylenko, S. Stirenko, Y. Gordienko, Improvement of Image Compression Performance by Deep Neural Networks, IEEE 19th International Conference on Smart Technologies (EUROCON-2021), Lviv, Ukraine.

Ключові слова. Нейронні мережі, глибока компресія зображень, машинне навчання, квантизація змінних на рівні каналу, авто-кодувальники.

ABSTRACT

for a master's thesis

made on the topic:

Image compression method based on neural network

by student: Vasylenko Dmytro Yevheniiovych

Master's Thesis: The study consists of an introduction and four sections. Total workload is: 86 sheets of body text, 42 illustrations, 12 tables. 42 various sources were used.

The urgency of the problem. With the growth of the network technologies amount of the produced and processed data in the life by each of us is growing with an enormous speed, it was found out that data produced in the last 20 years by the humanity is larger than that was created since its dawn, thus creating for us the task to find an effective method for the image compression and storage is the one most prioritised of all.

An image, being one of the most important types of the data, has to have an effective approach for transfer and storage giving humanity an unbelievable opportunity. Yet for the last years we used only the classic compression algorithms, thus creation of the new state of art neural network mechanism for the image compression may set the new stage in the growth of the compression technologies.

The purpose and objectives of the study. The purpose of the masters research is a creation and development of characteristics improvement methods of a neural network based deep image compression mechanism.

To reach the goals that were set it was fulfilled and completed the next tasks:

- research the definition of the image and its structure with the existing image formats;
- investigate deeply the existing compression method lossless or lossy;

- research the existing methods of the deep image compression with usage of the neural networks;
- analysed and improved the existing method of the deep image compression based on the channel level variable quantization;
- illustrated the models work and analysed the results of the created improved variant of the deep image compression framework;

Object of the study. Method of the deep image compression based on the neural network.

Subject of study. Effectiveness of the deep image compressions method work and ways of its improvement

Research methods and Scientific Novelty. To succeed in the goals that were set by the masters research it was used the machine learning and neural network training methods. Scientific novelty is being guaranteed by the next bullets below:

- it was proposed to implement the novel neural network block that is a part of a state of art encoder for deep image compression network;
- implemented the programme product of the neural network for succeeding in the task
- training was done on the cloud provider - Microsoft Azure

Personal contribution of the applicant. The master's research is an independently performed work, which reflects the personal author's approach and personally obtained theoretical and applied results related to the problem of deep image compression with usage of the neural networks. The formulation of the purpose and objectives of the study was carried out jointly with the supervisor.

Practical value. Received results may be freely used in the future studies done by the next topics:

- machine learning
- deep image compression

- compression done with the neural networks

Publications. Results of the research that was done in scope of the masters thesis was presented on the next conferences and published in the proceedings of these conferences:

- D. Vasylenko, S. Stirenko, Y. Gordienko, Deep Image Compression System for the Better Rate-Distortion Performance, The International Conference on Security, Fault Tolerance, Intelligence (ICSFTI2021 Online), Kyiv, Ukraine.
- D. Vasylenko, S. Stirenko, Y. Gordienko, Improvement of Image Compression Performance by Deep Neural Networks, IEEE 19th International Conference on Smart Technologies (EUROCON-2021), Lviv, Ukraine.

Keywords. Neural Networks, Deep Image Compression, Machine Learning, Channel Level Variable Quantization, Autoencoders.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI (англ. artificial intelligence) - штучний інтелект.

BMP (англ. bitmap picture) - формат збереження растрових даних.

BPP (англ. bits per pixel) - метрика якості процесу стискання зображення що відповідає за кількість бітів у піксель зображення.

CNN (англ. convolutional neural network) - згорткова нейронна мережа.

CPU (англ. central processing unit) - центральний процесор.

DCT (англ. discrete cosine transform) - дискретна косинусна трансформація.

Dec (англ. decoder) - декодувальник.

DWT (англ. discrete wavelet transform) - дискретна імпульсна трансформація.

EBCOT (англ. embedded block coding with optimized truncation) - кодування вбудованого блоку с оптимізованим процесом усічення.

EM (англ. expectation-maximization) - алгоритм максимізації очікування.

Enc (англ. encoder) - кодувальник.

FTP (англ. file transfer protocol) - протокол передачі даних по мережі інтернет.

GMM (англ. gaussian mixture models) - метод суміші Гаусових розподілень.

GPU (англ. graphics processing unit) - графічний процесор, в більшості випадків використовується для рендерингу зображення, у машинному навчанні виконує обчислення що потребують значної кількості ядер.

GRU (англ. gated recurrent units) - контролюємий рекурентний блок.

IoT (англ. internet of things) - інтернет речей.

LZW (англ. Lempel-Ziv-Welch) - універсальний алгоритм стиснення даних без втрат.

ML (англ. machine learning) - машинне навчання.

MS-SSIM (англ. multiscale structure similarity) - метрика якості зображення що відповідає за структурну схожість стисненого зображення відносно оригінального на більшому масштабі.

PNG (англ. portable network graphics) - формат збереження графічної інформації з використанням алгоритму стиснення без втрат Deflate.

PSNR (англ. peak signal to noise ratio) - метрика якості зображення що відповідає за коефіцієнт сигналу відносно шуму на зображенні.

QUA (англ. quantizer) - квантувальник

RBF (англ. radial basis function) - функція з набору однотипних радіальних даних що використовується у машинному навчанні

ReLU (англ. rectified linear unit) - активаційна функція нейронної мережі

RLE (англ. run-length encoding) - кодування довжин серій

RNN (англ. recurrent neural network) - рекурентна нейронна мережа.

SABR (англ. spatially adaptive bit rate) - просторово адаптивний біт рейт

SE (англ. squeeze and excitation) - метод стискання-збудження блоку

SSIM (англ. structure similarity) - метрика якості зображення що відповідає за структурну схожість стисненого зображення відносно оригінального.

STDEV (англ. standard deviation) - середньоквадратичне відхилення

TPU (англ. tensor processing unit) - тензорний процесор

ВСТУП

Розвиток інтернет технологій та розповсюдження смартфонів досягло неймовірного темпу за останні два десятиліття. За статистикою майже 3.5 мільярди людей у світі постійно користуються інтернетом одночасно. Запуск проєкту StarLink призведе до розповсюдження безпровідного високошвидкісного інтернет покриття по всьому світу до 2025 року. Проте незважаючи на досить стрімкий розвиток технологій з кожним роком об’єм даних продовжує зростати з неймовірною швидкістю.

Задля підтримання високої якості досвіду користувача є однією із основних проблем розробників програмного забезпечення двадцять першого тисячоліття. Через нестабільне інтернет підключення та не статичну швидкість мобільного інтернет покриття розробники намагаються максимально зменшити об’єм переданих даних. Проте процес стиснення окрім неймовірної кількості позитивних результатів призводить до втрати якості даних. У випадку із музикою це втрата якості та частоти. У випадку із зображенням це втрата таких характеристик як: колір, чіткість; поява артефактів на зображенні. Майже всі провідні технологічні компанії вкладають значну кількість коштів у створення кращих механізмів стиснення зображення із найменшими втратами якості.

Проте на даний момент всі найпопулярніші механізми компресії зображення використовують детерміновані алгоритми, деякі з них краще виконують дану функцію, деякі гірше. Основною ідеєю даної магістерської дисертації є повний відхід від класичного підходу до компресії даних. Оскільки останні двадцять років показали нам наскільки нейронні мережі є ефективними у рішенні найрізноманітніших проблем, таких як : передбачення фондового ринку, системи класифікатори, розпізнавання об’єктів, генерація музики, високоефективне підвищення якості зображення. Тому було обрано дослідити використання нейронних мереж у проблемі компресії. Одним із потенційних рішень було обрано інтегрування нейронної мережі у процес квантування зображення.

Тема даної магістерської дисертації є надзвичайно інноваційною та актуальною, адже знайдення нового більш ефективного методу компресії відкриває нові горизонти у даній галузі та надає можливість значно покращити якість користування всіма інтернет сервісами простим користувачам. Оптимізація компресії може повністю змінити наш досвід користування інтернетом та значно пришвидшити розвиток IoT сфери у світі.

РОЗДІЛ 1

ПРОБЛЕМА СТИСНЕННЯ ЗОБРАЖЕННЯ

1.1 Основи стиснення даних

Стиснення - це процес більш оптимізації кодування даних для зменшення розміру файлу. Один із доступних типів стиснення називається стисненням без втрат. Це означає, що стиснений файл буде відновлений точно до вихідного стану без втрати даних під час процесу розпакування. Це важливо для стиснення даних, адже у випадку якщо файл був пошкоджений він стає непридатним для використання оскільки дані будуть втрачені. Інша категорія стиснення - це стиснення з втратами, яке часто використовується в мультимедійних файлах для стиснення музичний файлів і зображень, при якому дані видаляються.

Алгоритми стиснення без втрат використовують методи статистичного моделювання для зменшення кількості повторюваної інформації в файлі. Деякі з методів можуть включати видалення пробілів, уявлення рядків символів, що повторюються одним машинним символом або заміну тих символів, що повторюються меншими бітовими послідовностями. Стиснення файлів дає багато переваг під час оптимізації використання пам’яті для збереження інформації. Базова схема зображена на Рис. 1.1.

При стисненні, кількість бітів, використовуваних для зберігання інформації, зменшується. Файли меншого розміру скорочують час передачі, коли вони передаються через мережу Інтернет. Стиснуті файли також займають менше місця для зберігання. Компресія файлів дозволяє стиснути кілька невеликих файлів в один для більш зручної передачі по електронній пошті.

Оскільки стиснення - це складний з математичної точки зору процес, він може потребувати багато часу, особливо коли задіяна велика кількість файлів. Деякі алгоритми компресії також пропонують різні рівні стиснення, при цьому більш високі, забезпечують менший розмір файлу, але вимагають більшого часу на виконання. Це системно дуже інтенсивний процес, що вимагає цінних

ресурсів, що іноді може призводити до помилок через нестачу оперативної пам’яті. При такій великій кількості варіантів алгоритмів стиснення у користувача, що завантажує стислий файл, може не бути необхідною програми для його розпакування.

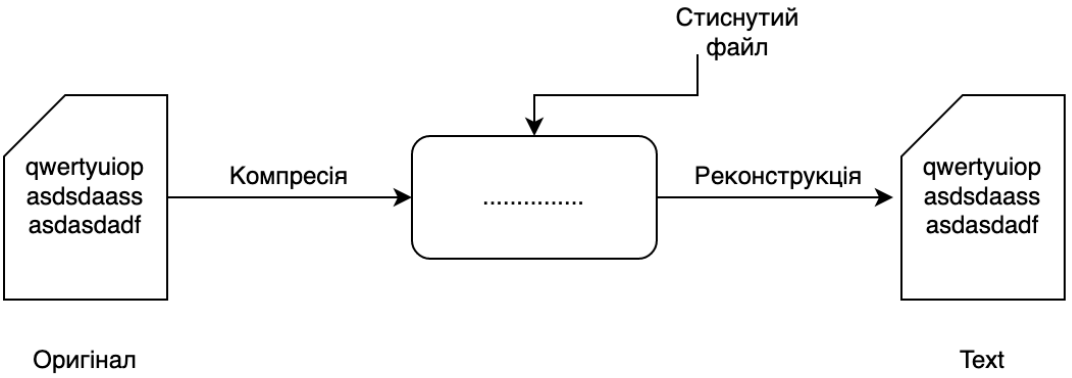

Рис 1.1 Високорівнева схема процесу компресії

Деякі протоколи передачі можуть включати в себе необов'язкове вбудоване стиснення (наприклад, FTP має параметр стиснення MODE-Z), так що час на процес компресії даних іншим процесом перед передачею може звести нанівець деякі переваги використання такої опції в протоколі (оскільки результат процесу може бути настільки незначним, що витрати часу на нього будуть не ефективними та призведуть до де-оптимізації протоколу).

Цілком можливо, що «зовнішнє» стиснення в наші дні більш ефективно, і що будь-яка опція стиснення в протоколі, ймовірно, повинна бути застарілою. Однак також не слід забувати, що вбудоване стиснення дає більш швидкі загальні результати, але, зазвичай, лише з великими файлами або навпаки. Тому у процесі імплементації слід проводити експерименти, щоб з'ясовувати, який із факторів найбільш важливий для користувача.

1.1.1 Поняття зображення

У сучасній комп'ютерній графіці є два типи зображень. Растрові і векторні зображені на Рис.1.2.

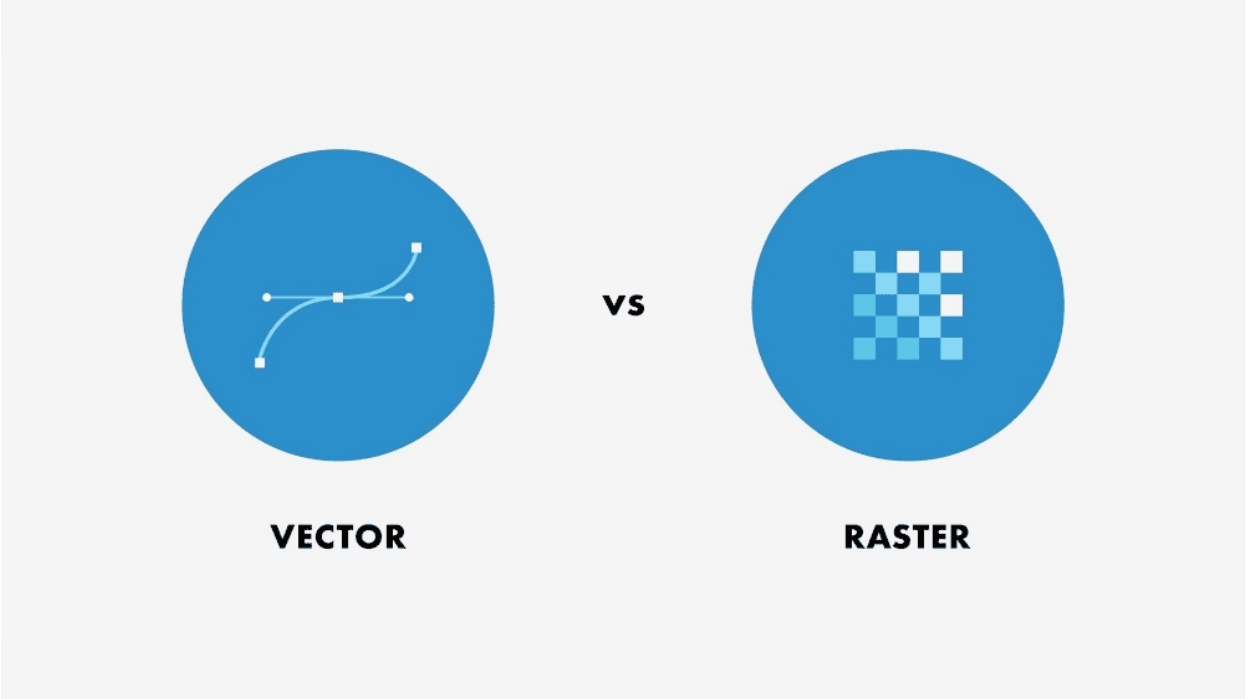

Рис 1.2 - Основні типи зображень[1]

Але перш ніж ми почнемо з того, що таке векторна графіка і растрова графіка, ми повинні зрозуміти кілька основних термінів:

Піксель: в комп'ютерній графіці піксель, точки або елемент зображення - це фізична точка зображення. Піксель - це просто найменший адресний елемент зображення, представленого на екрані.

Більшість зображень, які ми бачимо на екрані нашого комп'ютера, є растровими зображеннями. Зображення складається з набору пікселів, так званих растрових зображень.

Растрове зображення: в комп'ютерній графіці - це відображення деякого домену (наприклад, діапазону цілих чисел) в біти, тобто значення, рівні нулю або одиниці. Він також називається двійкового масивом або індексом растрового зображення. Більш загальний термін растрове зображення відноситься до карти пікселів, де кожен може зберігати більше двох кольорів, таким чином, використовуючи більше одного біта на піксель. Часто для цього також використовується растрове зображення. У деяких контекстах термін растрове зображення має на увазі один біт на піксель, в той час як растрове зображення використовується для зображень з кількома бітами на піксель.

У растрових зображеннях для зберігання інформації використовуються растрові зображення. Це означає, що для великого файлу потрібна велика растрове зображення. Чим більше зображення, тим більше місця на диску займає файл зображення. Наприклад, зображення 640 x 480 вимагає, щоб інформація була збережена для 307 200 пікселів, в той час як зображення 3072 x 2048 (з 6,3-мегапіксельною цифровою камери) повинно зберігати інформацію для колосальних 6291456 пікселів. Ми використовуємо алгоритми, які стискають зображення, щоб зменшити ці розміри файлів. Формати зображень, такі як jpeg і gif, є поширеними найпоширенішими для стиснутих зображень. Зменшити ці зображення легко, але збільшення растрового зображення робить його піксельним або просто розмитим. Отже, для зображень, які необхідно масштабувати до різних розмірів, ми використовуємо векторну графіку.

Оскільки дані зображення засновані на пікселях, вони залежать від роздільної здатності. Кількість пікселів, що становлять зображення, а також кількість цих пікселів, що відображаються на дюйм, визначають якість зображення. Як ви вже здогадалися, чим більше пікселів в зображенні і чим вище розширення, тим вище якість зображення.

Наприклад, якщо ми масштабуємо растрове зображення, щоб збільшити його без зміни розширення, воно втратить якість і буде виглядати розмитим або піксельним. Це тому, що ми розтягуємо пікселі на більшій площі, роблячи їх менш різкими. Це звичайна проблема, але її можна вирішити за допомогою програм редагування растрових зображень, таких як Photoshop, для зміни дозволу і правильного масштабування зображень. Структура зображена на Рис. 1.3.

Розширення файлів: .BMP, .TIF, .GIF, .JPG.

Використання послідовних команд, математичних операторів або програм, які розміщують лінії або фігури в дво- або тривимірному середовищі, називається векторною графікою. Векторна графіка найкраще підходить для друку, оскільки вона складається з ряду математичних кривих. В результаті

векторна графіка друкується чітко навіть при збільшенні. У фізиці: вектор - це те, що має величину і напрямок.

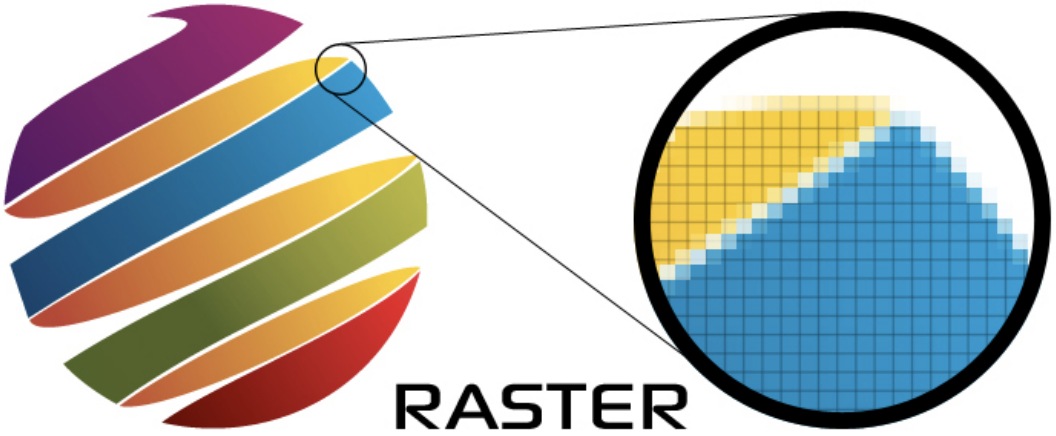

Рис. 1.3 - Структура растрового зображення[2]

У векторній графіці файл створюється і зберігається як послідовність векторних операторів. Замість того, щоб мати в файлі біт для кожного біта малювання лінії, ми використовуємо команди, які описують серії точок, які необхідно з'єднати. В результаті виходить файл набагато меншого розміру. Замість того, щоб намагатися відслідковувати мільйони крихітних пікселів в растровому зображенні, векторних зображеннях або штрихових малюнках, відстежуйте точки і рівняння для ліній, які їх з'єднують. Загалом, векторні зображення складаються з контурів або штрихових малюнків, які можна нескінченно масштабувати, оскільки вони працюють на основі алгоритмів, а не пікселів.

Одна з особливостей векторних зображень полягає в тому, що можливо змінювати їх розмір нескінченно більше або менше, і вони все одно будуть друкуватися так само чітко, без збільшення (або зменшення) розміру файлу. Якщо ви згадаєте шкільну геометрію, рівняння для кола з центром (h, k) і радіусом r буде $(x - h) * 2 + (y - k) * 2 = r * 2$. Якщо ви хочете зробити коло більше, ви просто збільшуєте значення r - замість того, щоб відстежувати велику кількість пікселів, системі просто потрібно відстежувати одне число. Це майже не займає місця в файлі.

Отже, які типи графіки зазвичай бувають векторними? Що ж, майже всі файли комп'ютерних шрифтів засновані на векторних зображеннях букв - тому

їх можна масштабувати, але при цьому літери будуть чіткими. У всіх малюнках Microsoft Office використовується векторна графіка, а більшість діаграм і графіків, що створюються Office або програмним забезпеченням для статистичного аналізу, є векторними. Зазвичай векторні зображення створюються в таких додатках, як Adobe Illustrator або CorelDRAW. Векторні ілюстрації відмінно підходять для логотипів, ілюстрацій / ілюстрацій, анімації і тексту. Структура зображена на Рис. 1.4.

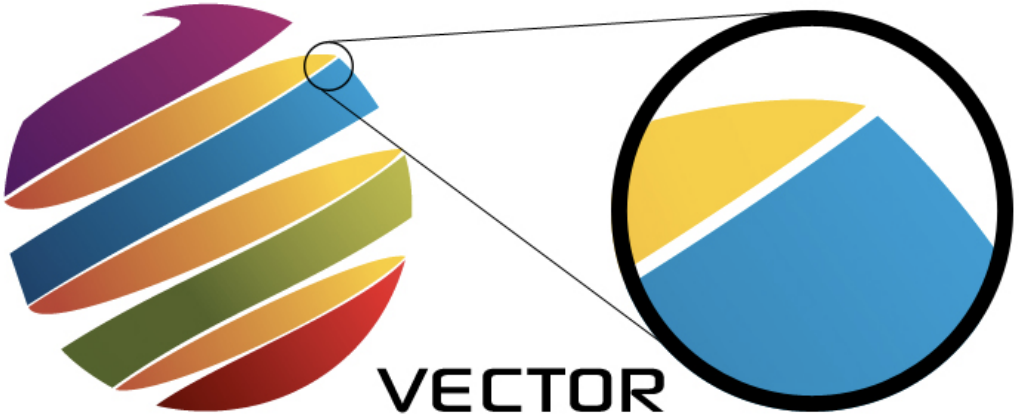

Рис. 1.4 - Структура векторного зображення[3]

Розширення файлів: .SVG, .EPS, .PDF, .AI, .DXF.

Їх можна перетворити навпаки:

- Вектор в растр.

Принтери і пристрої відображення є растровими. В результаті нам потрібно перетворити векторні зображення в растровій формат, перш ніж їх можна буде використовувати, тобто відображати або друкувати. Необхідне розширення грає життєво важливу роль у визначенні розміру створюваного растрового файлу. Тут важливо зазначити, що розмір конвертованого векторного зображення завжди залишається незмінним. Векторний файл зручно конвертувати в ряд форматів растрових файлів, але піти протилежним шляхом складніше (бо іноді нам потрібно редагувати зображення при перетворенні з растра в вектор)

- Растр в вектор.

Трасування зображень в обчислювальній техніці можна віднести до векторизації, і це просто перетворення растрових зображень в векторні зображення. Цікавим застосуванням векторизації є оновлення зображень і відновлення роботи. Векторизацию можна використовувати для вилучення інформації, яку ми втратили. Paint в Microsoft Windows створює вихідний файл растрового зображення. В Paint легко помітити нерівні лінії. При такому перетворенні розмір зображення різко зменшується. В результаті в цьому сценарії точне перетворення неможливо. Через різних наближень і редагування, які виробляються в процесі конвертації, перетворенні зображення не хорошої якості.

Основна відмінність між векторної і растрової графікою полягає в тому, що растрова графіка складається з пікселів, а векторна графіка складається з контурів. Растрова графіка, така як gif або jpeg, являє собою масив пікселів різних кольорів, які разом утворюють зображення.

1.1.2 Стиснення зображень

По суті, компресія зображень - це коли процес видалення або групування певних частин файлу зображення, задля зменшення його розміру. Прикладами застосування є:

- Для оптимізації сайту. Сайти з не стиснутими зображеннями можуть завантажуватися довше і через це можуть відмовлятися від відвідувачів.
- Для відправки та завантаження зображень. Завантаження не стиснутого зображення може зайняти деякий час, а деякі поштові сервери мають обмеження на розмір файлу.
- Для зменшення навантаження на жорсткий диск.

1.1.3 Типи стиснення зображень

Виділяють два типи методів стиснення зображень - без втрат та з втратами.

1.1.3.1 Стиснення з втратами

Щоб зробити зображення ще меншого розміру, стиснення з втратами видаляє деякі частини фотографії. Однак це не означає, що фотографія буде погано виглядати. Ось два основних типи стиснення з втратами.

JPG також відомий як JPEG, цей формат позбавляє від фрагментів фотографії, які ви можете помітити в залежності від застосовуваного рівня стиснення. Нормальна ступінь стиснення буде непомітна, в той час як надмірне стиснення може бути очевидним. Є й інші способи зниження якості зображення JPG. Якщо виконати операцію обертання до певного кута на форматі JPG, то можна помітити різницю в якості. Це пов'язано з тим, що зображення повинне стискатися при кожному повороті, втрачаючи при цьому деякі дані. Однак існують програми, які обертають JPG без втрат. Така ж деградація застосовується при перезбережжні JPG кілька разів.

Термін JPEG є абревіатурою від Joint Photographic Experts Group, яка створила стандарт. Дуже часто звертаючись до зображень збережених у файлі в форматі JPEG, ми насправді маємо на увазі оболонку JFIF (формат обміну файлами JPEG) отож для більш глибокого розуміння слід звернути увагу на базу структуру самого зображення.

Сенсор камери покривається масивом кольорових фільтрів (CFA), оj зазвичай є фільтром Байєра, який складається з мозаїчної матриці розміром 2x2 зі червоних, зелених, синіх і зелених фільтрів. Зелені фотосенсори є елементами, чутливими до яскравості, а червоний і синій - чутливими до кольоровості. Байєр використовував у два рази більше зелених елементів, ніж червоних або синіх, щоб імітувати фізіологію людського ока. З CFA ми отримуємо дані зображення одного кольору на піксель. Це не підходить для стиснення JPEG. Ці дані зображення реконструюються для отримання триплетів кольорів RGB на піксель зображення. Отриманий таким чином необроблений файл зображення являє собою растрове зображення (2D-масив). Він дуже великий за розміром.

Отже, нам потрібно стиснути цей файл, і саме тут JPEG вступає в гру. Даний формат перш за все - це метод стиснення з втратами, що означає, що він використовує наближення і часткове відкидання даних для стиснення вмісту. Отже, це є незворотнім. Стиснення JPEG засноване на наступних двох спостереженнях:

- Спостереження №1: людські очі не так добре бачать колір , як ми яскравість.
- Спостереження №2: людське око не може розпізнати високочастотні зміни інтенсивності зображення.

GIF стискає файли, зменшуючи кількість наявних кольорів. Якщо фотографія містить більше 256 кольорів (максимальна кількість, яку могли мати старі комп'ютери), цей формат зробить зображення менш привабливим. Найкраще використовувати GIF-файли для анімованих зображень.

1.1.3.2 Стиснення без втрат

Стиснення без втрат - це метод, який використовується для зменшення розміру файлу при збереженні тієї ж якості, що і до його стиснення. Наприклад, в камері DSLR є можливість зберігати фотографії в форматі RAW або JPEG. Файли RAW не мають стиснення і відмінно підходять, якщо ви професійний редактор фотографій. Але вони займають більше місця. JPEG, з іншого боку, не заповнить ваш жорсткий диск так швидко, але деякі дані будуть втрачені при перетворенні.

Типи зображень без втрат включають:

RAW - зустрічається в багатьох дзеркальних фотокамерах і зберігає всі дані про висвітлення, отримані з сенсора камери. Для професіонала це чудова новина. Однак ці типи файлів, як правило, досить великі за розміром. Крім того, існують різні версії RAW, і вам може знадобитися деякий програмне забезпечення для редагування файлів.

PNG - стискає зображення, щоб зберегти їх невеликий розмір, шукаючи візерунки на фотографії і стискаючи їх разом. Стиснення є оборотним, тому після відкриття файлу PNG зображення повністю відновлюється.

BMP - формат, призначений виключно для Microsoft. Без втрат, але використовується не часто.

1.2 Поняття нейронної мережі

Термін нейронні мережі історично відноситься до мереж нейронів в головному мозку ссавців. Нейрони є його основними обчислювальними

одиницями, і вони пов'язані один з одним в мережі для обробки даних. Це може бути дуже складним завданням, і тому динаміка таких нейронних мереж у відповідь на зовнішні стимули часто буває досить складною.

Входи і виходи кожного нейрона змінюються в залежності від часу у вигляді ланцюжків піків, але також і сама мережа з часом змінюється: ми дізнаємося і покращуємо наші можливості обробки даних, встановлюючи нові зв'язки між нейронами. натхненний архітектурою і динамікою нейронів мозку. В алгоритмах використовуються ідеалізовані моделі нейронів.

Проте, фундаментальний принцип той же: штучні нейронні мережі навчаються, змінюючи зв'язку між своїми нейронами. Такі мережі можуть виконувати безліч завдань по обробці інформації. Нейронні мережі можуть, наприклад, навчитися розпізнавати структури в наборі «навчальних» даних. і до деякої міри узагальнюють те, що вони дізналися. Навчальний набір містить список вхідних шаблонів разом зі списком відповідних позначок або цільових значень, які кодують властивості вхідних шаблонів, які мережа повинна вивчити.

Штучні нейронні мережі можуть бути навчені дуже точно класифікувати такі дані, регулюючи силу зв'язку між їх нейронами, і можуть навчитися узагальнювати результат на інші набори даних - за умови, що нові дані не надто відрізняються від даних навчання. Яскравим прикладом проблеми цього типу є розпізнавання об'єктів на зображеннях, наприклад в послідовності зображень з камери, зроблених безпілотним автомобілем. Недавній інтерес до машинного навчання з нейронними мережами частково викликаний успіхом нейронних мереж в розпізнаванні візуальних об'єктів.

Ще одна задача, в якій нейронні мережі процвітають, - це машинний переклад з динамічними або рекурентними мережами. Такі мережі приймають пропозиції в якості вхідних даних. Послідовно передаючи слово за словом, мережа виводить слова з перекладеного пропозиції. Рекурентні мережі можна ефективно навчати на великих навчальних наборах вхідних пропозицій і їх перекладів. Google Translate працює таким чином. Рекурентні мережі також зі

значним успіхом використовувалися для передбачення хаотичної динаміки. Все це приклади контрольованого навчання, коли мережі навчаються пов'язувати певні цілі або ярлики з кожним входом.

Штучні нейронні мережі також гарні для аналізу великих наборів немаркованих, часто багатовимірних даних, де може бути важко визначити апріорі, які питання є найбільш актуальними та корисними. Алгоритми неконтрольованого навчання організовують немарковані вхідні дані по-різному: вони можуть, наприклад, виявляти знайомство і схожість (кластери) вхідних шаблонів та інших структур у вхідних даних.

Алгоритми неконтрольованого навчання добре працюють, коли є надмірність вхідних даних, і вони особливо корисні для великих, багатовимірних наборів даних, де може бути проблемою виявити кластери або інші структури даних шляхом перевірки. дві крайності навчання з учителем і без вчителя. Подумайте, як агент може навчитися орієнтуватися в складній навколишньому середовищу, щоб якомога швидше переміщатися з одного місця в інше або якомога швидше витрачаючи трохи енергії. Метод навчання з підкріпленням дозволяє агенту робити саме це, оптимізуючи свою поведінку у відповідь на сигнали навколишнього середовища у вигляді штрафів і винагород. Коротше кажучи, агент вчиться діяти таким чином, що отримує позитивний зворотний зв'язок (винагороду) частіше, ніж штраф. Інструменти для машинного навчання з нейронними мережами були розроблені давно, більшість з них - у другій половині минулого століття.

У 1943 році Маккалок і Піттс проаналізували[4], як мережі нейронів можуть обробляти інформацію. Використовуючи абстрактну модель нейрона, вони продемонстрували, як такі блоки можуть бути пов'язані один з одним для представлення логічних функцій. Їх аналіз і висновки сформульовані з використанням логічного синтаксису Карнапа, а не в термінах алгебри, як ми звикли сьогодні. Проте, їх модель нейрона - це, по суті, бінарна порогова одиниця, тісно пов'язана з фундаментальним будівельним блоком більшості нейромережевих алгоритмів машинного навчання на сьогоднішній день. Тому в

цій книзі ми називаємо цю модель нейроном Маккаллок-Піттса. Метою цього раннього дослідження нейронних мереж було пояснення нейрофізіологічних механізмів.

Можливо, найбільш значним досягненням став принцип навчання Хебба, що описує, як нейронні мережі навчаються, зміцнюючи зв'язки між нейронами, які активні одночасно. Цей принцип описаний в книзі Хебба «Організація поведінки: нейропсихологічна теорія»[5], опублікованій в 1949 р. Приблизно десять років по тому дослідження штучних нейронних мереж активізувалися, чому сприяла впливова робота Розенблатта[6].

У 1958 році він сформулював правило навчання нейрона Маккаллока-Піттса, пов'язане з правилом Хебба, і продемонстрував, що правило сходиться до правильного рішення для всіх завдань, які може вирішити ця модель.

Він ввів термін перцептрону для багаторівневих мереж нейронів Мак-Каллока-Питтса і показав, що такі нейронні мережі в принципі можуть вирішувати завдання, які не може виконувати один нейрон Маккаллок-Піттса. Однак в той час не існувало загального правила навчання для перцептронів. У роботах Мінскі і Пейперті [7] особлива увага приділялася геометричним аспектам навчання. Це дозволило їм довести, які проблеми можуть вирішувати перцептрони, а які ні. У 1969 році вони узагальнили ці результати в своїй книзі «Перцептрони». Введення в обчислювальну геометрію.

1.2.1 Базові компоненти нейронної мережі

1.2.1.1 Нейрони

У штучних нейронних мережах способи обробки інформації та передачі сигналів надзвичайно ідеалізовані. Мак-Калок і Піттс змоделювали нейрон, обчислювальну одиницю нейронної мережі, як бінарну порогову одиницю. У нього є тільки два можливих виходу або стану: активний і неактивний. Рис. 1.5 зображує фізичну модель нейрона яка є основою для математичної.

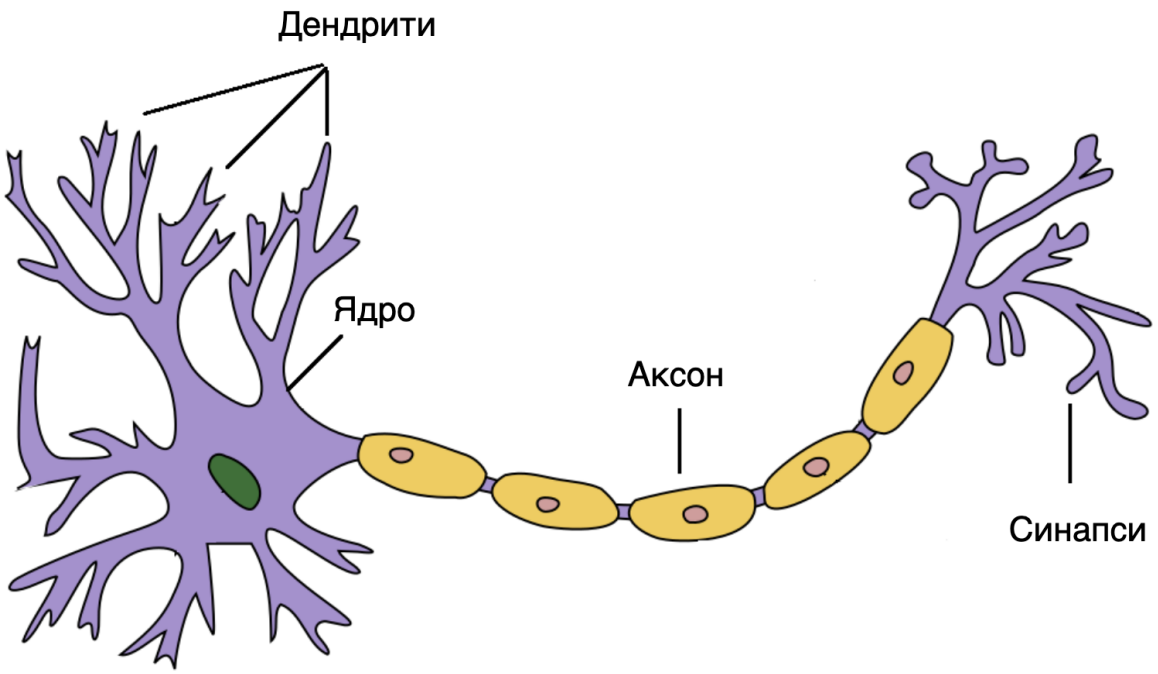

Рис. 1.5 - Фізична модель нейрону

Для обчислення вихідних даних пристрій підсумовує зважені вхідні дані. Якщо сума перевищує заданий поріг, стан нейрона вважається активним, в іншому випадку - неактивним.

Проілюстрована більш загальна модель, ніж вихідна. Вона виконує багаторазові обчислення з дискретним кроком по часу $s_t = 0, 1, 2, 3\ldots$. Стан нейрона номер j на часовому кроці позначається як:

$$s_j(t) = \{-1, \text{неактивний}; 1 \text{ активний}\}$$

З огляду на стани $s_j(t)$, нейрон з номером i розраховує:

$$s_i(t+1) = sgn(\sum_{j=1}^{N} w_{ij} s_j(t) - \Theta_i) \equiv sgn[\, b_i(t)\,]$$

На Рис. 1.6 індекс нейрона - i, він отримує вхідні дані від N нейронів. Сила зв'язку нейрона j з нейроном і позначається w_{ij}. Граничне значення для нейрона i позначено Θ_i. Індекс $t = 1, 2, 3, 4$ ....позначає дискретну тимчасову послідовність кроків обчислення, а $sgn(b)$ позначає функцію знака. Тут $sgn(b)$ - Сігнум-функція

$$sgn(b) = \{-1,\ b < 0\,;\ 1,\ b > 0\}$$

Аргумент Сігнум-функції,

$$b_i(t) = \sum_{j=1}^{N} w_{ij} s_j(t) - \theta_i$$

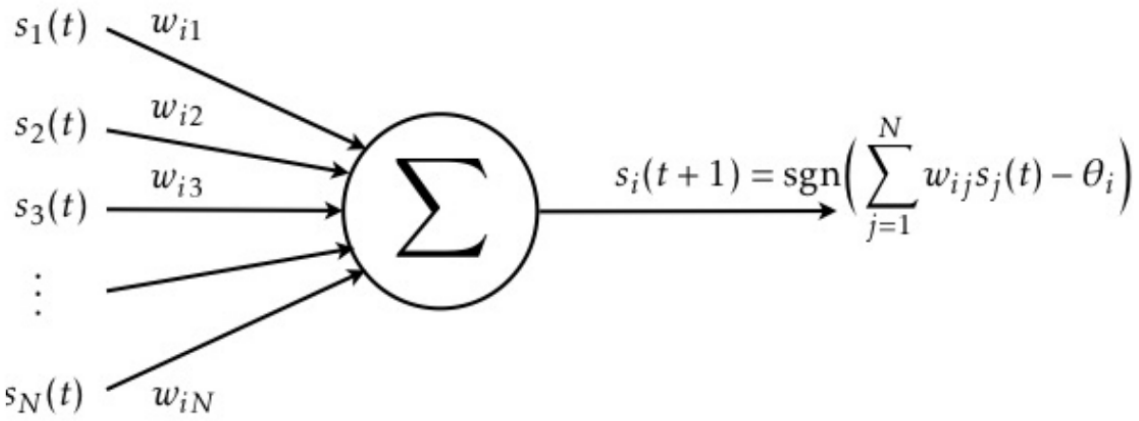

Рис. 1.6 - Принципова схема нейрона Мак-Каллока-Піттса.

називається локальним полем. Ми бачимо, що нейрон виконує середньозважене значення вхідних даних $s_j(t)$. Параметри w_{ij} називаються вагами. Тут перший індекс i відноситься до нейрона, який[8] виконує обчислення, а j позначає все нейрони, які з'єднуються з нейроном i. Загалом, ваги між різними парами нейронів беруть різні числові значення, що відображають різну силу синаптичних зв'язків.

Ваги можуть бути позитивними або негативними, і ми говоримо, що при $w_{ij} = 0$зв'язку немає. У цій роботі ми посилаємося на модель, описану як нейрон Маккаллок-Піттса, хоча їх вихідна модель мала деякі додаткові обмеження на ваги. Поріг 1 для нейронів позначений θ_i. Нарешті, обчислення виконується для всіх нейронів паралельно, і виходи S_i є входами для всіх нейронів на наступному часовому кроці. Отже, виходи мають аргумент часу t + 1. Ці кроки повторюються багато разів, що призводить до тимчасового ряду рівнів активності всіх нейронів в мережі.

1.2.1.2 Функції активації

Модель Маккаллок-Піттса апроксимує моделі піків активності в двох станах, -1 та +1, що представляють не активний і активний періоди. Для

багатьох обчислювальних задач цього достатньо, і для наших цілей не має значення, що динаміка електричних сигналів в корі сильно відрізняється в деталях. Зрештою, мета полягає не в тому, щоб моделювати нейронну динаміку в головному мозку, а в тому, щоб побудувати обчислювальні моделі, натхненні реальною нейронною динамікою. Очевидно, що найпростіша модель, описана вище, повинна бути в деякій мірі узагальнена для певних завдань і питань. Наприклад, стрибок Сігнум-функції при $b = 0$ може викликати великі коливання рівнів активності мережі нейронів, викликані нескінченно малими змінами локальних полів при $b = 0$. Щоб послабити цей ефект, можна дозволити нейрону постійно реагувати на його входи, замінюючи

$$s_i(t + 1) = g\left(\sum_j w_{ij} s_j(t) - \theta_i\right)$$

Тут $g(b)$ - безперервна функція активації. Це може бути просто лінійна функція $g(b) \infty b$. Але ми бачимо, що багато завдань вимагають нелінійних функцій активації, таких як $tanh\,(b)$. Коли функція активації є безперервною, нейронні стани також беруть безперервні значення, а не тільки дискретні значення -1 і +1. В якості альтернативи можна використовувати частково-лінійну функцію активації. Частково це мотивується моделлю для зв'язку між електричним струмом що проходить через клітинну мембрану в нейрон, і мембранним яким можна відобразити значення потенціалу U. У найпростіших моделях мембранний потенціал відображає нейрон як конденсатор. У нейроні з не герметичним інтегратором та функції активації витік додається резистором R, паралельним конденсатору С, так що

$$I = \frac{U}{R} + C\frac{dU}{dt}$$

Для постійного струму мембранний потенціал зростає від нуля як функція часу, $U(t) = R\,I[1 - exp(-\,t/\tau)]$, де $\tau = R_C$- постійна часу моделі. нейрон виробляє як пік, коли мембранний потенціал перевищує критичне значення U_c. Відразу після цього мембранний потенціал встановлюється на нуль (і знову починає рости). У цій моделі швидкість стрільби $f(I)$, таким чином,

визначається як t^{-1}_{c}, де t_c - рішення $U(t) = U_c$. Звідси випливає, що швидкість має граничне значення. Іншими словами, система працює як випрямляч:

$$f(I) = \left\{0,\ I \leq U_c / R;\ \left[\tau log(\frac{RI}{RI - U_c})\right]^{-1}, I \geq U_c / R\right\}$$

Головне, що є поріг, нижче якого результат строго дорівнює нулю а функція активації якісно виглядає як частково-лінійна функція

1.2.2 Основні типи нейронних мереж

На даний момент існує велика кількість типів нейронних мереж що мають різну спеціалізацію. Ми виділимо наступні основні типи:

- Нейронна мережа з прямим зв'язком - штучний нейрон

Ця нейронна мережа - одна з найпростіших форм, в якій дані або вхідні дані переміщаються в одному напрямку. Вони проходять через вхідні вузли і виходять на вихідні вузли. Ця нейронна мережа може мати або не мати приховані шари. Простіше кажучи, вона має переднє та не має зворотного поширення, зазвичай з використанням кваліфікуючої функції активації.

На Рис. 1.7 представлена одношарова мережу з прямим зв'язком. Тут сума творів входів і ваг обчислюється і подається на вихід. Вихід вважається, якщо він вище певного значення, тобто порогу (зазвичай 0), і нейрон спрацьовує з активованим виходом (зазвичай 1), і якщо він не спрацьовує, видається деактивовано значення (зазвичай -1).

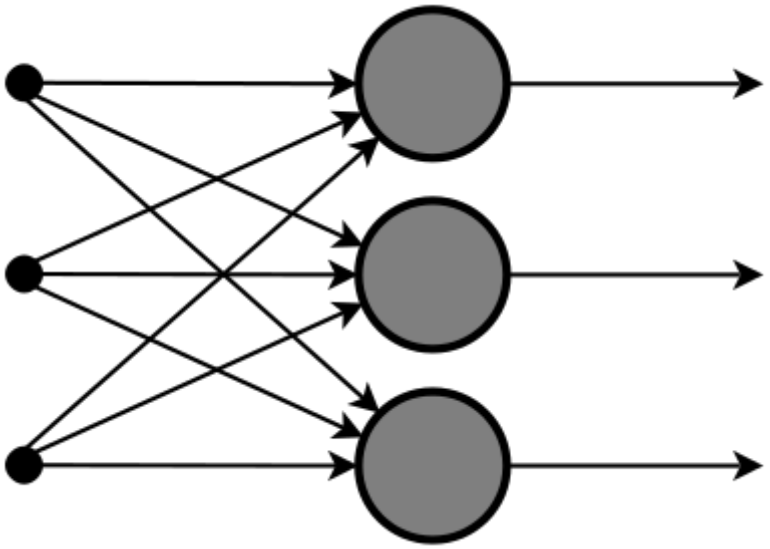

Рис. 1.7 - Модель нейронної мережі з прямим зв’язком

Нейронні мережі з прямим зв'язком застосовуються в комп'ютерному зорі і розпізнаванні мови, де класифікація цільових класів ускладнена. Такі нейронні мережі реагують на дані з високим значенням шуму і прості в обслуговуванні.

- Радіальна базисна функція Нейронна мережа

Радіальні базисні функції враховують відстань від точки до центру. Функції RBF мають два рівня: спочатку функції об'єднуються з радіальної базисної функцією на внутрішньому рівні, а потім вихідні дані цих функцій беруться до уваги при обчисленні того ж виходу на наступному часовому кроці, який, по суті, є пам'яттю.

Нижче приведена діаграма, на якій показано відстань, яка розраховується від центру до точки на площині, аналогічної радіусу кола. Тут міра відстані, яка використовується в евклідової системі, також можуть використовуватися інші заходи відстані. Модель залежить від максимального охоплення або радіусу кола при класифікації точок по різних категоріях. Якщо точка знаходиться всередині або навколо радіуса, ймовірність того, що нова точка буде віднесена до цього класу, висока. При переході від одного регіону до іншого може відбуватися перехід, і цим можна управляти за допомогою бета-функції.

Ця нейронна мережа була застосована в системах відновлення живлення. Адже енергосистеми збільшуються в розмірах і складніше. Обидва ці чинники збільшують ризик серйозних відключень електроенергії. Після відключення електроенергії необхідно якомога швидше і надійніше відновити подачу електроенергії. Відновлення живлення зазвичай відбувається в наступному порядку:

- ❑ Першим пріоритетом є відновлення електропостачання основних споживачів в спільнотах. Ці клієнти надають послуги з охорони здоров'я і безпеки для всіх, і відновлення харчування в першу чергу дозволяє їм допомогти багатьом іншим. Основні клієнти включають медичні установи, шкільні ради, важливу муніципальну інфраструктуру, а також поліцію і пожежні служби.

- ❏ Потім зосереджуються на основних лініях електропередач і підстанціях, які обслуговують більшу кількість клієнтів.
- ❏ Приділяють більш високий пріоритет ремонту, який дозволить якомога швидше повернути до обслуговування найбільшу кількість клієнтів.
- ❏ Потім відновлюється електропостачання невеликих кварталів, приватних будинків і підприємств.

- Самоорганізована нейронна мережа Кохонена

Метою карти Кохонена є введення векторів довільної розмірності в дискретну карту, що складається з нейронів. Карту необхідно навчити для створення власної організації навчальних даних. Вона складається з одного або двох вимірів. При навчанні карти розташування нейрона залишається постійним, але ваги розрізняються залежно від значення. Цей процес самоорганізації складається з різних частин, на першому етапі кожне значення нейрона инициализируется невеликою вагою і вхідним вектором.

На другому етапі нейрон, найближчий до точки, є «нейроном-переможцем», і нейрони, підключені до нейрона-переможця, також будуть рухатися до точки, як показано на малюнку нижче. Відстань між точкою і нейронами розраховується по евклидову віддалі, нейрон з найменшою відстанню перемагає. В ході ітерацій всі крапки об'єднуються в кластери, і кожен нейрон представляє кожен тип кластера. В цьому суть організації нейронної мережі Кохонена.

Нейронна мережа Кохонена зображена на Рис. 1.8 використовується для розпізнавання закономірностей в даних. Його можна знайти в медичному аналізі для кластеризації даних за різними категоріями. Карта Кохонена дозволила з високою точністю класифікувати пацієнтів з гломерулярними і тубулярна формами. Ось докладне пояснення того, як він класифікується математично за допомогою алгоритму евклидова відстані. Нижче наведено зображення, на якому показано порівняння здорового і хворого клубочків.

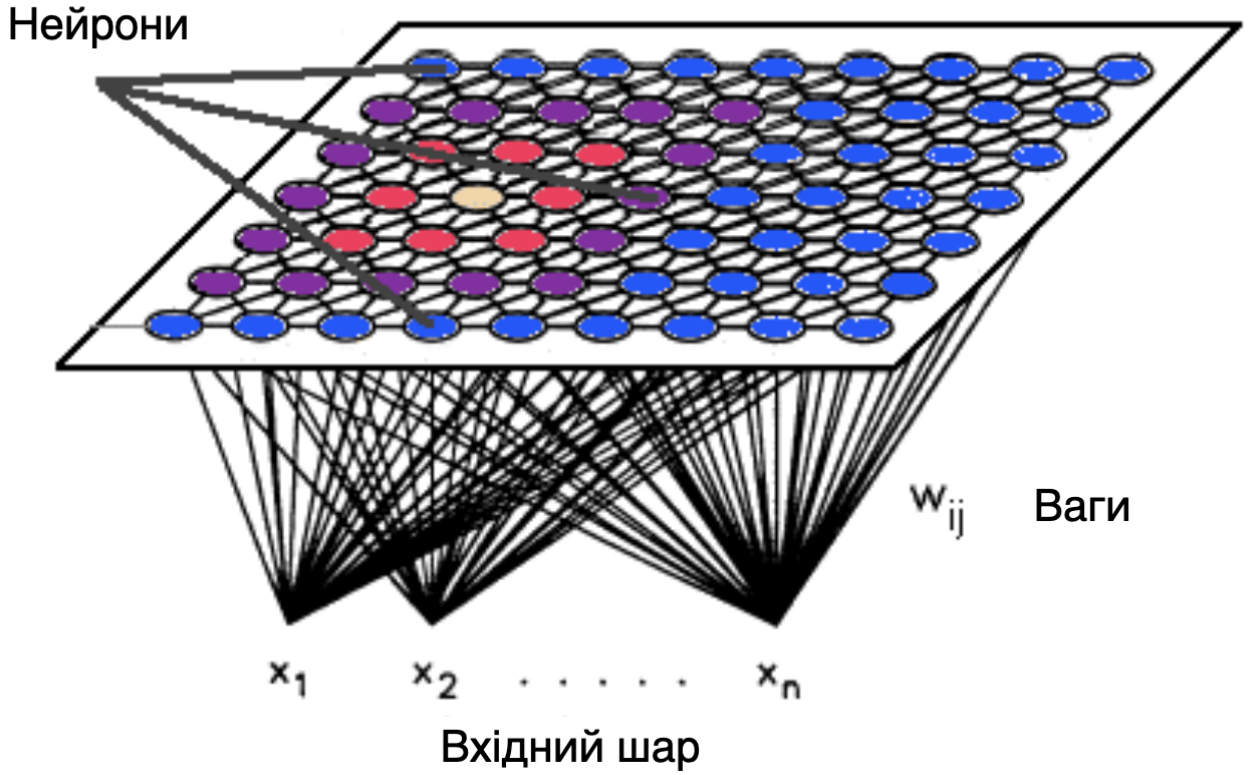

Рис. 1.8 - Схема мережі Кохонена

- Рекурентна нейронна мережа (RNN) - з довгостроковою - короткостроковою пам'яттю

Рекурентна нейронна мережа працює за принципом збереження виводу шару і подачі його назад на вхід, щоб допомогти в прогнозуванні результату шару. Тут перший шар сформований аналогічно нейронної мережі з прямим зв'язком з добутку суми ваг і характеристик. Повторюваний процес нейронної мережі починається після його обчислення, це означає, що від одного тимчасового кроку до наступного кожен нейрон буде пам'ятати деяку інформацію, яку він мав на попередньому часовому кроці.

Це змушує кожен нейрон діяти як осередок пам'яті при виконанні обчислень. У цьому процесі нам потрібно дозволити нейронної мережі працювати над переднім поширенням і запам'ятати, яка інформація їй потрібна для подальшого використання. Тут, якщо прогноз невірний, ми використовуємо швидкість навчання або корекцію помилок, щоб внести невеликі зміни, щоб він

поступово працював в напрямку правильного прогнозу під час зворотного поширення.

Застосування рекурентних нейронних мереж можна знайти в моделях перетворення тексту в мову (TTS). У цьому дослідженні ми розглянемо мережу Deep Voice[9], яка була розроблена в лабораторії штучного інтелекту Baidu в Каліфорнії. Структура зображена на Рис. 1.9.

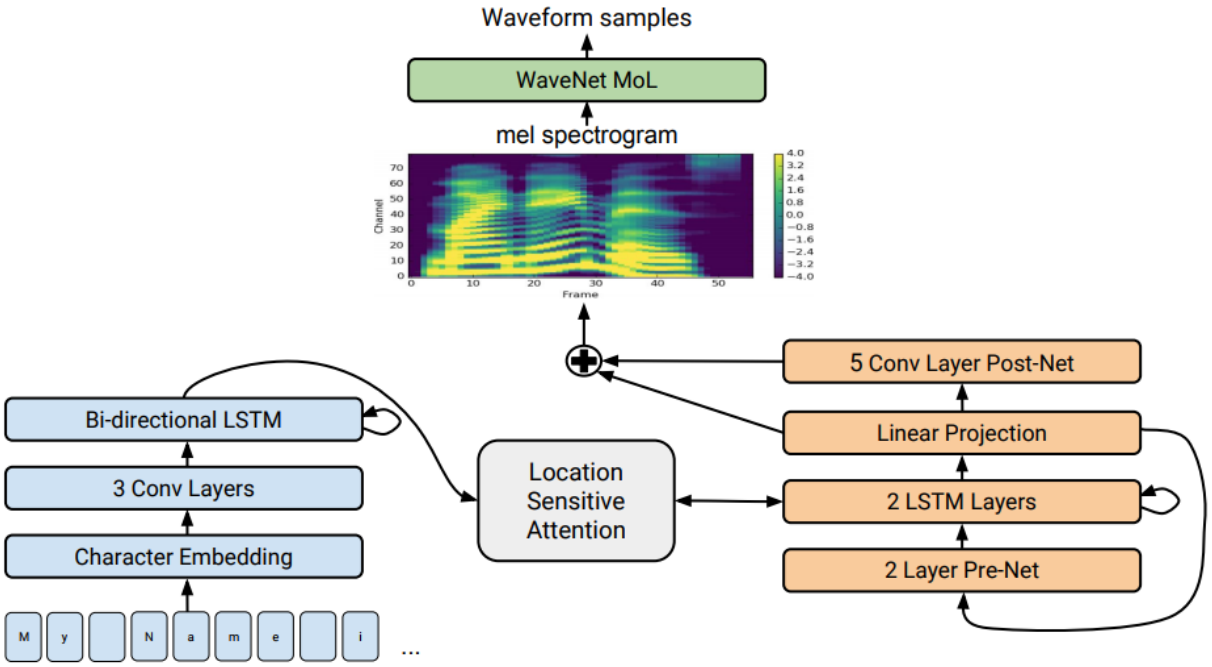

Рис. 1.9 - Схема роботи мережі Deep Voice[10]

Він був натхненний традиційною структурою перетворення тексту в мову, в якій всі компоненти були замінені нейронними мережами. Спочатку текст перетворюється в «фонему», а модель синтезу звуку перетворює його в мова.

- Згорткова нейронна мережа

Згорткові нейронні мережі схожі на нейронні мережі прямого поширення, в яких нейрони мають ваги і зміщення. Схема подібної мережі зображена на Рис. 1.10.

ЇЇ основне застосування в обробці сигналів і зображень, найпопулярнішою технологією наприклад є OpenCV в області комп'ютерного зору. Нижче представлено уявлення ConvNet, в цій нейронної мережі вхідні функції беруться пакетно, як фільтр.

Це допоможе мережі запам'ятовувати зображення по частинах і обчислювати над ними операції. Ці обчислення включають перетворення зображення з формату RGB або HSI в сірий колір. Як тільки ми отримаємо даний результат, зміни в значенні пікселя допоможуть виявити краї, і зображення можна буде класифікувати за різними категоріями.

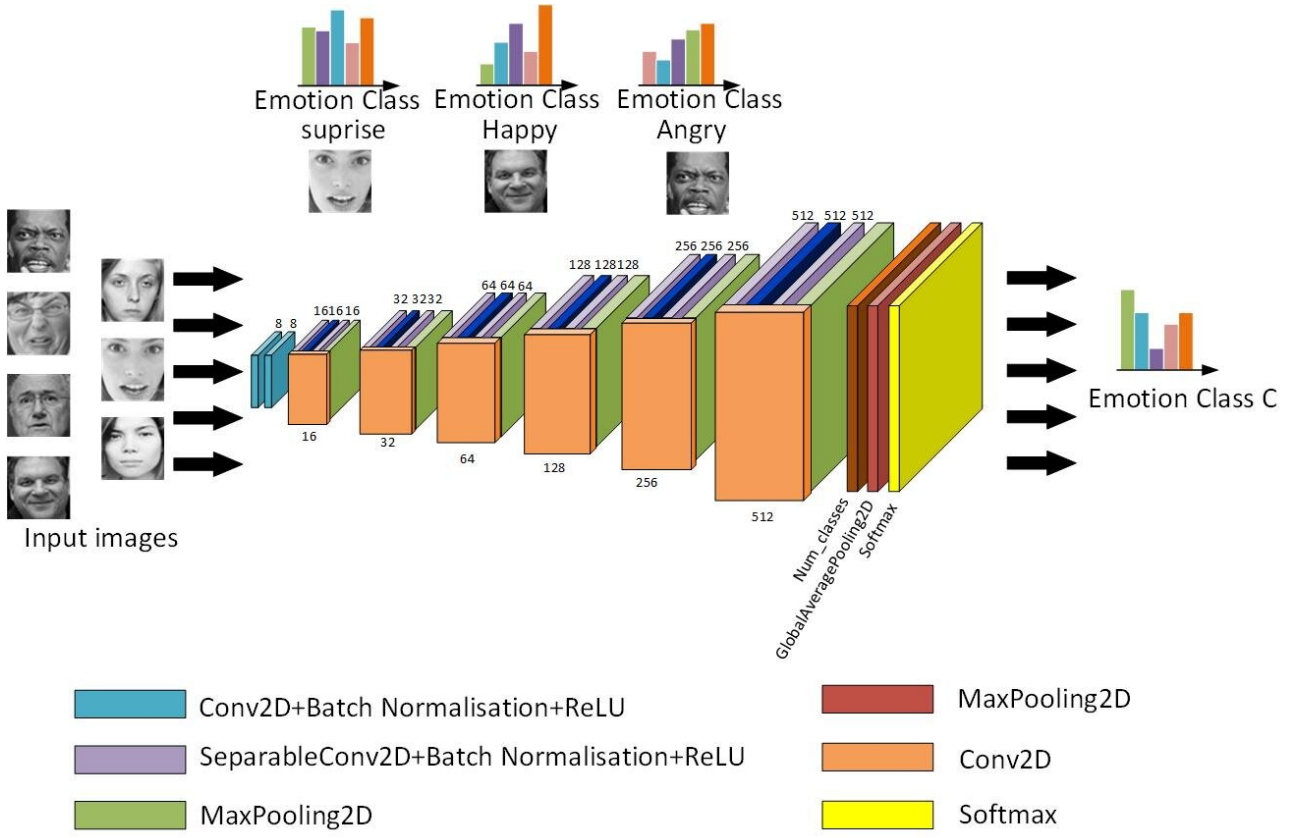

Рис. 1.10 - Схема роботи згорткової нейронної мережі[11]

Згорткова нейронна мережа застосовується в таких методах, як обробка сигналів і методи класифікації зображень. У методах комп'ютерного зору переважають згорткові нейронні мережі через їх точності класифікації зображень.

Впроваджується велика кількість методів аналізу проблем із серцем, легенями що базується на використанні нейронних мереж подібного типу, приклад імплементації подібного рішення зображено на Рис. 1.11.

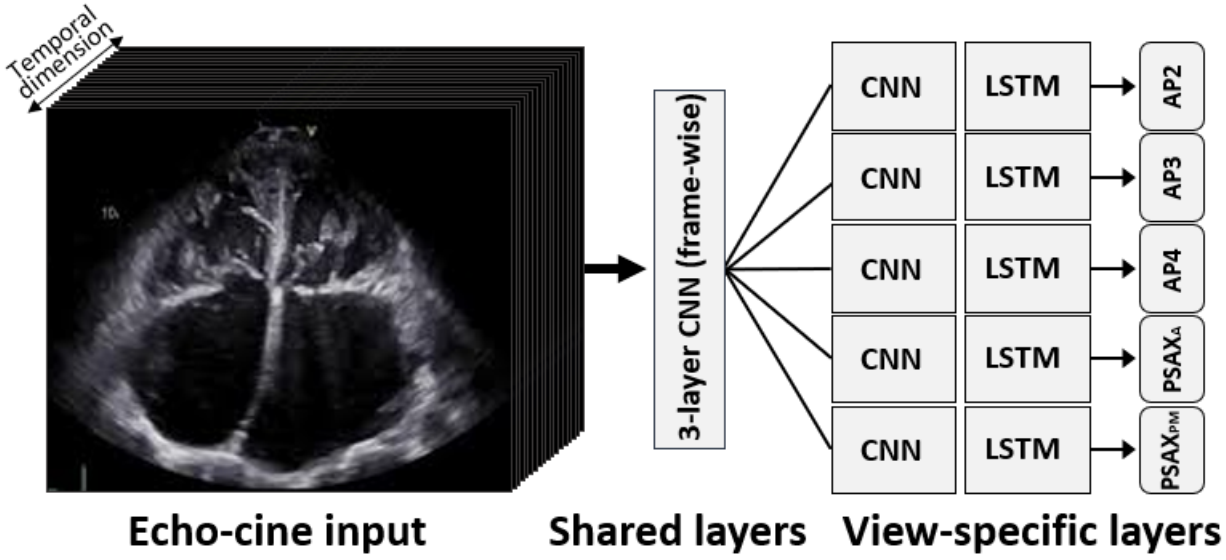

Рис. 1.11 - Приклад результати аналізу ехокардіограми[12]

- Модульна нейронна мережа

Модульні нейронні мережі - це набір різних мереж, що працюють незалежно і вносять свій внесок в результат.

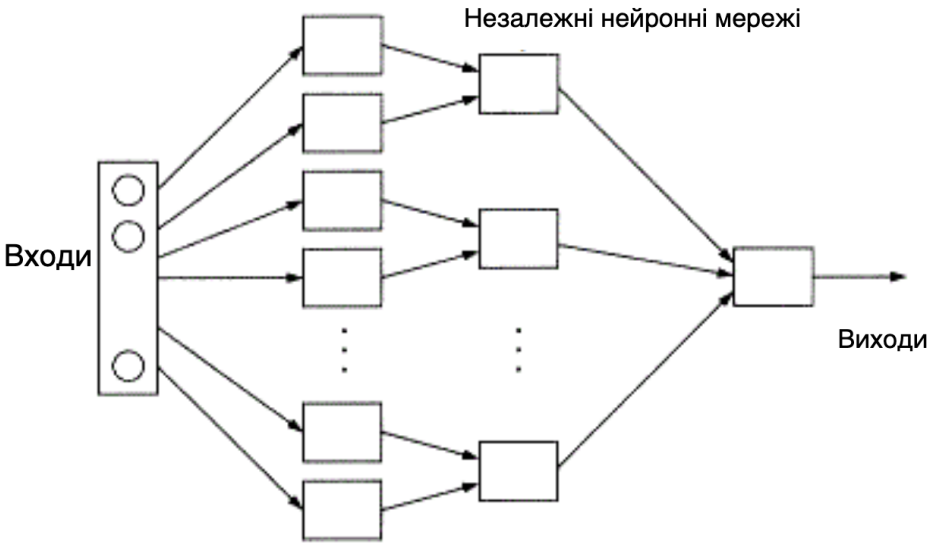

Рис. 1.12 - Схема модульної нейронної мережі

Кожна нейронна мережа має набір вхідних даних, які є унікальними в порівнянні з іншими мережами, що створюють і виконують підзадачі. Ці мережі, як на Рис. 1.12, не взаємодіють і не сигналізують один одному при виконанні завдань.

Перевага модульної нейронної мережі полягає в тому, що вона розбиває великий обчислювальний процес на більш дрібні компоненти, що знижує складність.

Ця розбивка допоможе зменшити кількість підключень і зведе нанівець взаємодія цих мереж один з одним, що, в свою чергу, збільшить швидкість обчислень. Однак час обробки буде залежати від кількості нейронів і їх участі в обчисленні результатів.

Продовження опису нейронних мереж в конкретному контексті теми дисертації буде надано відповідно в наступному розділ

ВИСНОВКИ ДО РОЗДІЛУ 1

Перший розділ було присвячено дослідженню поняття нейронної мережі її типам, видам форматів зображення. Була розглянута структура зображення у випадку комп'ютерної графіки, за для коректного складення картини сучасного стану розглянутої технології та надання можливості вирішення встановленою дисертацією задачею. В тому числі:

- Було проведено дослідження поточного стану технології обробки зображення, виокремлено основні типи зображень їх потенційні проблеми та різниці.
- Розглянута структурна різниця векторних та растрових зображень, що дозволяє глибше зрозуміти потенційне вирішення проблеми компресії цифрових зображень.
- Розглянута основна структура загальної нейронної мережі.
- Виокремлені проблеми переваги та недоліки сучасних типів нейронних мереж задля встановлення коректного типу для вирішення поставленою дисертацією проблеми.

Спираючись на проблеми виявлені під час проведеного аналізу та дослідження процесу компресії зображення, його форматів та типів стиснення, невід'ємними для вирішення поставленої дисертацією задачею є розв'язання наступних встановлених завдань:

- розробити спосіб компресії зображення що базується на використанні глибоко нейронної мережі у своєму ядрі
- розробити топологічну модель для нейронної мережі задля схематичної демонстрації створеної концепції
- розробити механізм підготовки тренувальних даних та знайдення механізму оцінки ефективності їх оцінки
- оцінити метрики ефективності запропонованого та розробленого методу, базуючись на них виділити переваги й недоліки, вказати на доменні області потенційно можливого його застосування.

РОЗДІЛ 2

МЕТОДИ КОМПРЕСІЇ ТА НЕЙРОННІ МЕРЕЖІ У СТИСНЕННІ ДАНИХ

2.1 Класичні алгоритми стиснення даних

Розглянемо алгоритм стиснення JPEG-101. Він складається з наступних кроків:

- Крок 1: Перетворення колірного простору RGB в YCbCr.

Кожен піксель вашого зображення зберігається як адитивна комбінація значень червоного, синього і зеленого кольорів. Кожне з цих значень може перебувати в діапазоні від 0 до 255. Ця модель позначення кольору називається моделлю RGB. Розглянемо піксель кольору хакі. Він буде збережений як (240, 230, 140). Слід брати до уваги спостереження №1 - яскравість важливіша для кінцевого якості сприйняття зображення, ніж колір. Таким чином, ми перетворимо колірний простір RGB в таке, в якому яскравість обмежена одним каналом. Це колірний простір називається YCbCr.

Тут Y - складова яскравості, а Cb, Cr - складові кольоровості. Це синя і червона різниця відповідно. Їх значення будуть в діапазоні від 0 до 255. Значення YCbCr можуть бути обчислені безпосередньо з RGB в такий спосіб:

- ❑ Y = 0,299 R + 0,587 G + 0,114 B
- ❑ Cb = - 0,1687 R - 0,3313 G + 0,5 B + 128
- ❑ Cr = 0,5 R - 0,4187 G - 0,0813 B + 128

- Крок 2: Понижуюча дискретизація

Оскільки кольоровість не дуже важлива, ми можемо зменшити дискретизацию і зменшити кількість кольору (компоненти CbCr). Як правило, колір зменшується в 2 рази в обох напрямках (вертикальному і горизонтальному), тобто Y вибирається в кожному пікселі, тоді як Cb і Cr вибираються в кожному блоці 2x2 пікселів. Тепер на кожні 4 пікселя Y буде існувати тільки 1 піксель CbCr. Ви не помітите значних змін в зображенні, але розмір файлу значно зменшилася. У програмах для редагування зображень вас зазвичай запитують, в якій якості ви хочете зберегти зображення. Фактично це

програма, яка запитує вас, скільки понижувальної дискретизації ви хочете зробити на зображенні.

- Крок 3: Дискретне косинусне перетворення

Кожен з трьох компонентів YCbCr стискається і кодується окремо використовуючи той же метод, який описаний тут. Поки розглянемо тільки один з цих компонентів. Решта компонентів обробляються так само.

  - базові зображення

DCT (discrete cosine transform) - це метод, який виражає кінцеву послідовність точок даних у вигляді суми косинусоїдальной функції. Для стиснення використовуються косинусні функції, а не синусоїдальні, через особливої різниці в їх граничному сходженні.

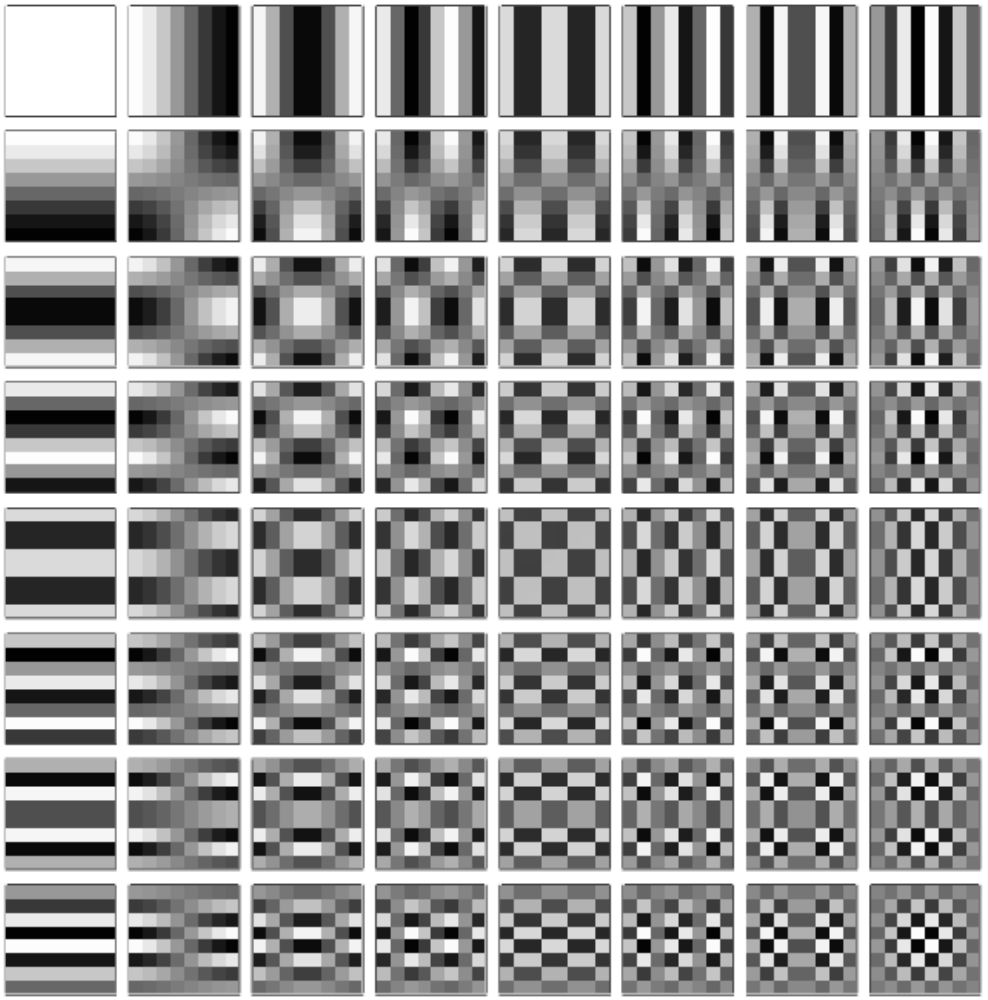

Рис. 2.1 - Результуючі зображення DCT методу[13]

Такі функції, як яскравість зображення, не повинні приймати нульові значення на кордоні, як це робить синус. Таким чином, такий сигнал складно апроксимувати лінійною комбінацією синусів.

Зображені на Рис. 2.1 64 базових зображення, побудовані з функцій косинуса на різних частотах по осях X і Y. Перше зображення, тобто базове - [0] [0], буде повністю білим, від зображення[0] [1] до [0] [7] ми можемо побачити, як частота збільшується по осі x. А з зображення [1] [0] до [7] [0], видно, що частота збільшується по осі Y. Останнє зображення [7] [7] буде повністю перевірене.

- суб-зображення

Всі зображення, які ми хочемо стиснути, діляться на фрагменти, кожний з яких має розмір 8x8 пікселів. Назвемо кожен з них другорядним зображенням. Цей фрагмент зображення може бути візуалізований як матриця розмірністю 8x8 пікселів. Ми збираємося стискати повне зображення по одному фрагменту за раз. Розглянемо приклад на Рис. 2.2. Значення даного компонента представлені в наступній матриці:

64	60	57	56	48	47	47	43
61	58	53	52	48	49	52	53
67	60	53	53	49	47	48	54
68	61	63	63	62	65	65	64
71	61	70	63	69	74	88	88
83	94	102	105	107	111	110	115
95	108	108	124	122	130	128	128
107	118	125	134	137	142	141	137

Рис. 2.2 - Приклад суб-зображення

Оскільки ми збираємося використовувати DCT, а косинусні хвилі змінюються від 1 до -1, ми збираємося центрувати наші значення навколо нуля. Це означає, що ми порушуємо діапазон з [0..255] на [-128..128]. Отже, ми віднімаємо 128 з кожного значення. Тепер наше вторинне зображення буде мати наступний вигляд на Рис. 2.3:

-64	-68	-71	-72	-80	-81	-81	-85
-67	-70	-75	-76	-80	-79	-76	-75
-61	-68	-75	-75	-79	-81	-80	-74
-60	-67	-65	-65	-66	-63	-63	-64
-57	-67	-58	-65	-59	-54	-40	-40
-45	-36	-26	-23	-21	-17	-18	-13
-33	-20	-20	-4	-6	2	0	0
-21	-10	-3	6	9	14	13	9

Рис. 2.3 - Зсунута матриця суб-зображення

В результаті ми маємо - додаткове зображення 8x8 для стиснення, 64 базових зображення. Наше завдання тут - перетворити фрагмент зображення в лінійну комбінацію цих 64 базових зображень. Часткове зображення може бути перетворено в цю виставу в частотної області з використанням нормалізованого двовимірного дискретного косинусного перетворення (DCT) типу II. Ми можемо думати про суб-зображенні, як про складений з зваженого набору цих 64 базових зображень, злитих друг над другом.

Отже, фрагмент зображення = C1f1 + C2f2 + C3f3 + ... C64f64, де Ci - деяка константа, а fi - базові зображення. Ми можемо знайти кожен з цих коефіцієнтів (Ci) за допомогою DCT (тип II). І отримуємо Рис. 2.4:

-376	-23	1	-2.5	-0.3	4	0.2	-2.6
-224	53	20	3.4	5	3	0.6	2.3
68	3.3	-14	-0.3	-2.8	-1.9	-4.7	-6.2
2.3	-8.9	-1.5	-3.8	-2.5	1.2	1.4	1.9
-8.4	1.2	1.9	3.3	-2.1	5	1.8	5.3
4.5	7.3	-7.4	1.9	1.3	-0.7	-1.5	-6
6.4	6.8	-3.2	-2.6	1.3	-2.1	1.7	1
-16	0.1	9	0.8	1.8	1.7	-1	1

Рис. 2.4 - Матриця суб-зображення після DCT

Це таблиця коефіцієнтів 8x8, яка представляє внесок кожного базового зображення у вторинне зображення.

- Крок 4: Квантування

Тепер ми проведемо операцію квантування над таблицею коефіцієнтів, яку ми отримали за допомогою DCT. Це справжня частина процесу з втратами.
У таблиці коефіцієнтів, які ми отримали через DCT, верхні ліві осередку відносяться до низькочастотної частини, а нижні праві осередки відносяться до високочастотної частини.

Ми знаємо, що високочастотні складові можуть бути усунені без значного погіршення зовнішнього вигляду зображення. (Спостереження №2) Отже, тепер ми готуємо таблицю квантування 8x8. У цій таблиці будуть дуже маленькі значення у верхній лівій частині і дуже високі значення в нижній правій частині.

Кожне значення в таблиці коефіцієнтів ділиться на відповідне значення в таблиці квантування і округляється до найближчого цілого числа. Тепер, через

високий подільника в правій нижній частині, розділені значення тут стають рівними нулю, тим самим усуваючи високочастотні дані.

Ця таблиця квантування залежить від кодувальника, і тому таблиця зберігається в заголовку зображення, щоб зображення можна було пізніше декодувати. Ось стандартна таблиця квантування JPEG на Рис. 2.5:

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Рис. 2.5 - Таблиця квантування

А ось наше вторинне зображення після квантування (після ділення кожного значення в нашій таблиці коефіцієнтів на відповідне значення в таблиці квантування) на Рис. 2.6.

Зверніть увагу, що в результуючій після квантування таблиці всі значення, крім верхнього лівого блоку 3x3, є нулями. Це дані з високою частотою, які ми видалили.

-24	-23	0	0	0	0	0	0
-19	4	1	0	0	0	0	0
5	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Рис. 2.6 - Результат квантування

- Крок 5: Кодування

Тепер у нас є стислий результат у вигляді 2D-масиву. Ми також знаємо, що багато хто з них - нулі. Отже, ми знайдемо кращий спосіб зберегти фрагмент зображення, ніж зберігати його у вигляді 2D-масиву. Ми будемо зберігати значення в зигзагоподібний порядку. Таким чином, дані будуть наступними: -24, -23, 19, 5, 4, 0, 0, 1, 0, 0, 0, 0, 1 з наступними 53 нулями.

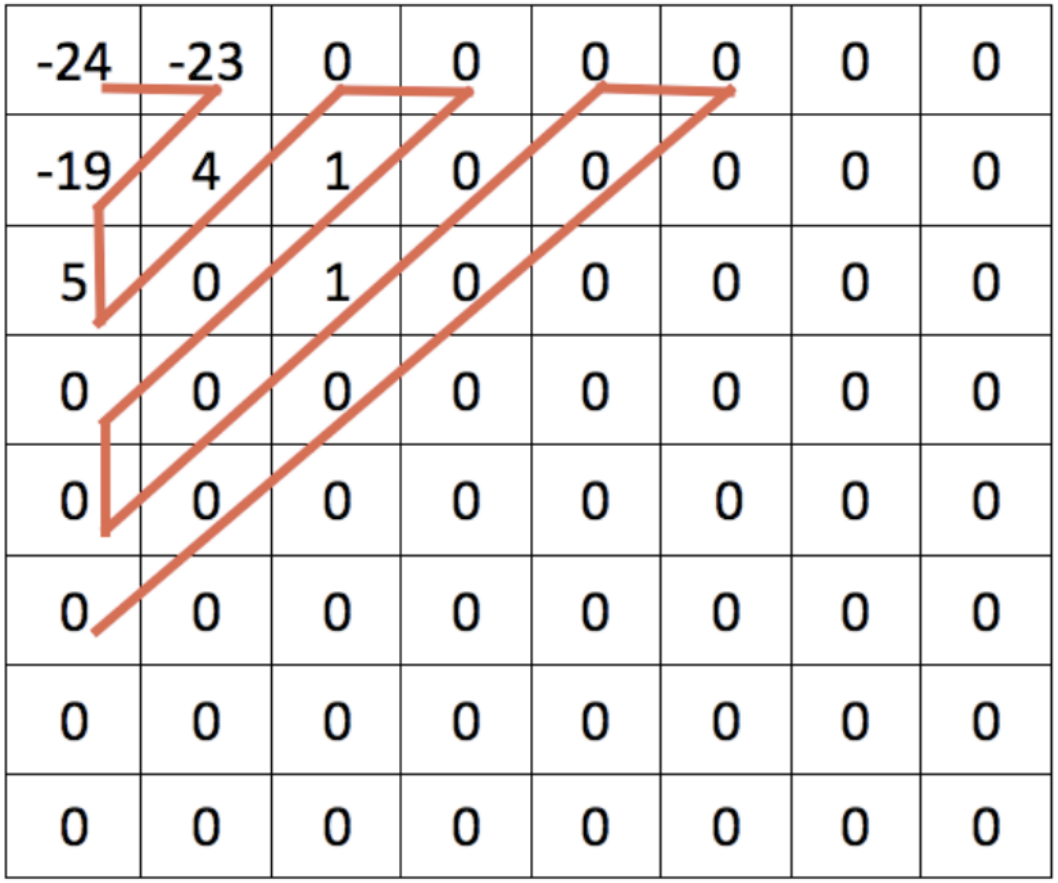

Рис. 2.7 - Схема зчитування матриці квантування

Дані цього шаблону на Рис. 2.7 можуть бути легко стиснуті за допомогою алгоритму RLE (run-length encoding) - кодування довжин серій. Кінцевий результат кодується з використанням комбінації кодування RLE і Huffman.

- Крок 6: Формування заголовку

Хоча файли JPEG / JFIF не мають формально певного заголовка, зазвичай він містить наступне:

- Маркер початку зображення (SOI) JPEG (0xFFD8)
- Маркери додатків
- Ширина в пікселях
- Висота в пікселях
- Кількість компонентів (наприклад, 3 для RGB)

Щоб розпакувати стиснений файл JPEG, просто зробіть зворотне. Використовуйте дискретне косинусне перетворення-III, щоб звернути DCT-II.

Давайте подивимося на один з найпопулярніших алгоритмів стиснення без втрат JPEG2000. Як і JPEG, стандарт стиснення зображень JPEG 2000 складається з чотирьох основних етапів алгоритму - попередньої обробки, перетворення, квантування і кодування. На відміну від JPEG, етап квантування є необов'язковим, якщо користувач бажає виконати стиснення без втрат. Там, де JPEG використовує розширену версію кодування Хаффмана, JPEG 2000 використовує новий метод кодування, званий вбудованим блоковим кодуванням з оптимізованим урізанням (EBCOT).

Крок 1 - Попередня обробка

Єдиний крок попередньої обробки, який ми будемо використовувати, - це центрування значень інтенсивності градацій сірого. З цією метою ми віднімаємо 127 з кожного значення інтенсивності в матриці зображення. Якби ми стискали кольорове зображення, ми б спочатку перетворили його в простір YCbCr, а потім центрованої б кожен з каналів Y, Cb і Cr.

Крок 2 - Перетворення

Одним з основних змін в стандарті JPEG2000 є використання DWT (discrete wavelet transform) дискретного вейвлет-перетворення замість DCT. Якщо ми виконуємо стиснення з втратами, ми використовуємо DWT з фільтром CDF97. Для стиснення без втрат ми використовуємо DWT в поєднанні з фільтром LeGall53 і виконуємо обчислення з використанням методу підйому, розробленого Вімом Свелденсом. В обох випадках ми обчислюємо 2-3 ітерації DWT. У нашому прикладі ми обчислюємо дві ітерації кожного перетворення.

Крок 3 - квантування

Квантований DWT. Повнорозмірна версія. Якщо ми виконуємо стиснення без втрат, DWT, кодується, і алгоритм завершується. Для стиснення з втратами JPEG2000 використовує схему квантування, частково аналогічну тій, що використовується в JPEG для блоків 8 x 8.

Для двох ітерацій DWT створює сім блоків, і кожен з цих блоків квантуется окремо. Значення в кожному блоці або переміщуються ближче до нуля, або конвертуються в нуль, а потім перетворюються в ціле число за

допомогою функції стану. Більш детальну інформацію про процес квантування можна знайти в підрозділі «Квантування». Результат для нашого поточного прикладу показаний праворуч.

Крок 4 - Кодування

На останньому етапі в стандарті стиснення ми використовуємо вбудоване блочне кодування з оптимізованим урізанням. Ми не розглядаємо тут метод EBCOT - embedded block coding with optimized truncation.

Для стиснення без втрат ми просто використовуємо EBCOT для кодування елементів вейвлет-перетворення, створеного за допомогою фільтра LeGall. У цьому випадку ми можемо зберегти зображення, використовуючи 215 544 біт. Початкове зображення в необробленому форматі вимагає 307 200 біт пам'яті, тому метод без втрат дає економію близько 30%. Ступінь стиснення - 5,6 біт на піксель.

Для стиснення з втратами EBCOT дозволяє нам використовувати тільки 84 504 біта замість вихідного розміру 160 * 240 * 8 = 307 200 біт. Ступінь стиснення складає близько 2,2 біт на піксель. Якщо ми стискаємо те ж зображення за допомогою JPEG, нам буде потрібно 103 944 біта для ступеня стиснення тобто 2,7 біт на піксель. Початкове зображення, зображення у форматі JPEG 2000 і зображення у форматі JPEG вказані нижче на Рис. 2.8.

Кодування Хаффмана. Кодування Хаффмана можна використовувати для стиснення всіх видів даних. Це ентропійний алгоритм, заснований на аналізі частоти символів в масиві.

Найбільш яскраво кодування Хаффмана може бути продемонстровано шляхом стиснення растрових зображень. Припустимо, у нас є растрове зображення 5 × 5 з 8-бітовим кольором, тобто 256 різних кольорів. Не стиснене зображення займає 5 x 5 x 8 = 200 біт.

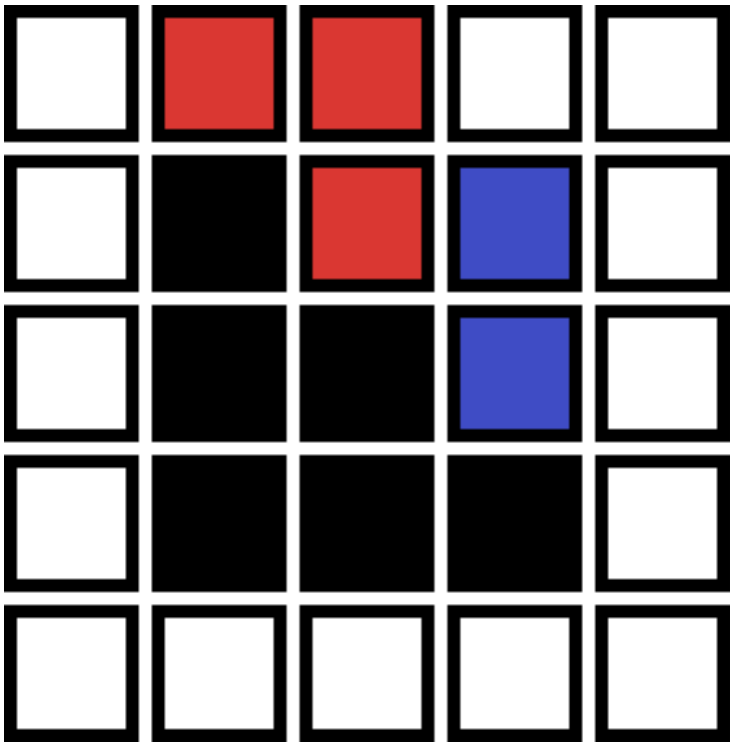

Рис. 2.8 - Схематичне представлення частини зображення у пікселях[14]

Спочатку ми підраховуємо, скільки разів кожен колір зустрічається в зображенні. Потім ми сортуємо кольору в порядку убування частоти. У підсумку ми отримуємо рядок, яка виглядає так як на Рис. 2.9:

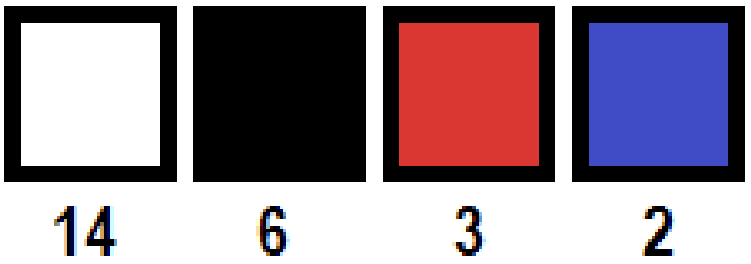

Рис. 2.9 - Схема результату сортування по кольорам[15]

Тепер ми з'єднуємо кольори, побудувавши дерево так, щоб найбільш віддалені від кореня, були найменш частими. Кольори з'єднані попарно, з'єднуючись утворюють вузол. Вузол може підключатися або до іншого вузла, або до кольору. У нашому прикладі дерево може виглядати так:

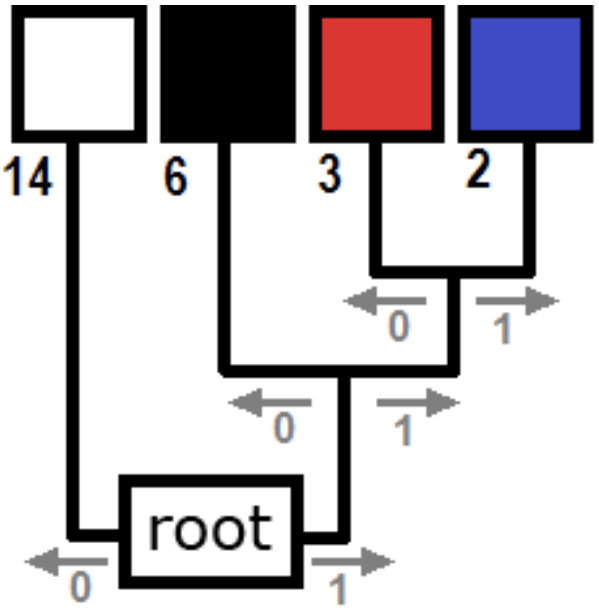

Рис. 2.10 - Дерево Хаффмана[16]

Наш результат відомий як дерево Хаффмана на Рис. 2.10. Його можна використовувати для кодування і декодування. Кожен колір кодується таким чином. Ми створюємо коди, переходячи від кореня дерева до кожного кольору. Якщо ми повернемо направо в вузлі, ми запишемо 1, а якщо повернемо наліво - 0. Цей процес дає кодову таблицю Хаффмана, в якій кожному символу призначається бітовий код, так що найбільш часто зустрічається символ має найкоротший код, в той час як найменш поширеному символу присвоюється найдовший код що видно з Рис. 2.11.

color	freq.	bit code
	14	0
	6	10
	3	110
	2	111

Рис. 2.11 -Таблиця кодування[17]

Створені нами дерево Хаффмана і кодова таблиця - не єдині можливі. Для нашого зображення можна створити альтернативне дерево Хаффмана, яке виглядає так як на Рис. 2.2:

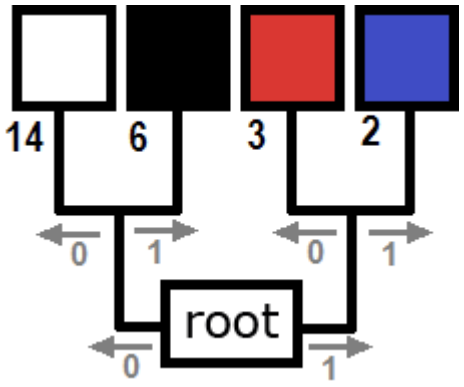

Рис. 2.12 - Другий варіант таблиці Хаффмана[18]

Тоді відповідна кодова таблиця буде на Рис. 2.13:

color	freq.	bit code
	14	00
	6	01
	3	10
	2	11

Рис. 2.13 - Другий варіант таблиці кодування[19]

У нашому прикладі переважніше використовувати перший варіант. Це тому, що він забезпечує краще стиснення для нашого конкретного зображення.

Оскільки кожен колір має унікальний бітовий код, який не є префіксом будь-якого іншого, кольори можуть бути замінені їх бітовими кодами в файлі зображення. Найбільш часто зустрічається колір, білий, буде представлений тільки одним бітом, а не 8 бітами. У чорних два біта. Червоний і синій займуть три. Після виконання цих замін 200-бітове зображення буде стиснене до 14 x 1 + 6 x 2 + 3 x 3 + 2 x 3 = 41 біт, що становить близько 5 байтів в порівнянні з 25 байтами в оригінальному документі.

Звичайно, для декодування зображення стислий файл повинен включати кодову таблицю, яка займає деякий місце. Кожен бітовий код, отриманий з дерева Хаффмана, однозначно визначає колір, тому при стисненні інформація не втрачається.

Цей метод стиснення широко використовується для кодування музики, зображень і деяких протоколів зв'язку. Зазвичай для підвищення ефективності використовується варіант алгоритму. Описаний метод зазвичай є частиною загальних алгоритмів стиснення, таких як Flate-ZIP для зображень або FLAC для музики.

При стисненні JPEG без втрат використовується алгоритм Хаффмана в чистому вигляді. JPEG без втрат широко використовується в медицині як частина стандарту DICOM, який підтримується основними виробниками медичного обладнання (для використання в ультразвукових апаратах, апаратах для отримання зображень ядерного резонансу, апаратах МРТ і електронних мікроскопах). Варіанти алгоритму JPEG без втрат також використовуються в форматі RAW, який популярний серед ентузіастів фотографії, оскільки він зберігає дані з датчика зображення камери без втрати інформації.

LZW (Lempel - Ziv - Welch) Техніка стиснення

Алгоритм LZW [20, 21] - дуже поширений метод стиснення. Цей алгоритм зазвичай використовується в GIF і, можливо, в PDF і TIFF. Команда Unix «стиснути», серед іншого, використовується. Це без втрат, тобто дані не втрачаються при стисненні. Алгоритм простий в реалізації і може забезпечити дуже високу пропускну здатність в апаратних реалізаціях. Це алгоритм широко використовується утиліти стиснення файлів Unix compress, який використовується в форматі зображень GIF.

Ідея належить на повторювані шаблони для економії місця для даних. LZW - це передовий метод стиснення даних загального призначення через його простоти і універсальності. Це основа багатьох утиліт для ПК, які заявляють, що «вдвічі збільшують місткість жорсткого диска».

Стиснення LZW працює шляхом читання послідовності символів, групування символів у рядки і перетворення рядків у коди. Оскільки коди займають менше місця, ніж рядки, які вони замінюють, ми отримуємо стиснення.

Стиснення LZW використовує кодову таблицю з 4096 кодів як загальний вибір для кількості записів таблиці. Коди 0-255 в кодової таблиці завжди призначаються для постановки окремих байтів з вхідного файлу. Коли кодування починається, кодова таблиця містить тільки перші 256 записів, а інша частина таблиці - порожні. Стиснення досягається за рахунок використання кодів з 256 по 4095 для подання послідовностей бітів.

У міру продовження кодування LZW ідентифікує повторювані послідовності в даних і додає їх в кодову таблицю. Декодування досягається шляхом взяття кожного коду з стисненого файлу і його перекладу через кодову таблицю, щоб визначити, який символ або символи він представляє.

Приклад: код ASCII. Зазвичай кожен символ зберігається з 8 двійковими бітами, що дозволяє використовувати до 256 унікальних символів для даних. Цей алгоритм намагається розширити бібліотеку до 9-12 біт на символ. Нові унікальні символи складаються з комбінацій символів, які раніше зустрічалися в рядку. Він не завжди добре стискається, особливо з короткими різноманітними струнами. Але він хороший для стиснення надлишкових даних і не вимагає збереження нового словника з даними: цей метод може як стискати, так і розпаковувати дані.

Ідея алгоритму стиснення полягає в наступному: у міру обробки вхідних даних в словнику зберігається відповідність між найдовшими словами що зустрічаються і списком кодових значень. Слова замінюються відповідними кодами, і тому вхідний файл стискається. Отже, ефективність алгоритму збільшується в міру збільшення кількості довгих повторюваних слів у вхідних даних.

Декомпресор LZW створює ту ж таблицю рядків під час розпакування. Він починається з перших 256 записів таблиці, ініціалізованих поодинокими

символами. Таблиця рядків оновлюється для кожного символу у вхідному потоці, крім першого. Декодування досягається шляхом зчитування кодів і їх трансляції через що будується кодову таблицю.

Переваги LZW перед Хаффманом:

- LZW не вимагає попередньої інформації про потік вхідних даних.
- LZW може стискати вхідний потік за один прохід.
- Ще одна перевага LZW - простота, що дозволяє швидко виконати.

2.2 Нейронні мережі та проблема стиснення даних

2.2.1 Кодувальник та декодувальник на основі рекурентної нейронної мережі

Глибина кожного шару відзначена над заднім кутом кожної площині. Ім'я та тип шару позначені як E_i: I / Hдля кодера (і D_j: I / Hдля декодера) всередині нижньої частини кожної площині. Згорткові ядра для введення мають розмір $I \times I$, а згорткові ядра для прихованого стану -$H \times H$. Неперіодичні шари з прямим зв'язком мають $H = 0$.

На вхід кодувальника надходить залишкове зображення: різниця між вихідним зображенням і реконструкцією попередньої ітерації. Для першої ітерації цей залишок є просто вихідне зображення. Перший і останній рівні в мережах кодера і декодера використовують згорткові блоки з прямим зв'язком ($H = 0$) з активацією $tanh$. Інші шари містять згорткові GRU(gated recurrent units) - стробовані рекурентні блоки. Що добре видно із Рис. 2.14.

Для забезпечення точного підрахунку бітової швидкості бінарізатор квантує свій вхід до ± 1. Це дасть нам нашу номінальну (до ентропійного кодування) швидкість передачі даних. З огляду на наш вибір частоти понижувальної дискретизації і глибини бінарізатора, кожна ітерація додає 18 біт на піксель до попередньої номінальної швидкості передачі даних. Просторовий контекст, який використовується кожним пікселем реконструкції, як функція або «бітових стеків» (тобто вихідних сигналів бінарізатора в одній просторової позиції), або вихідних пікселів зображення, може бути обчислена шляхом дослідження обрахування. пов'язані просторові опори кодера, декодера і векторів всіх станів.

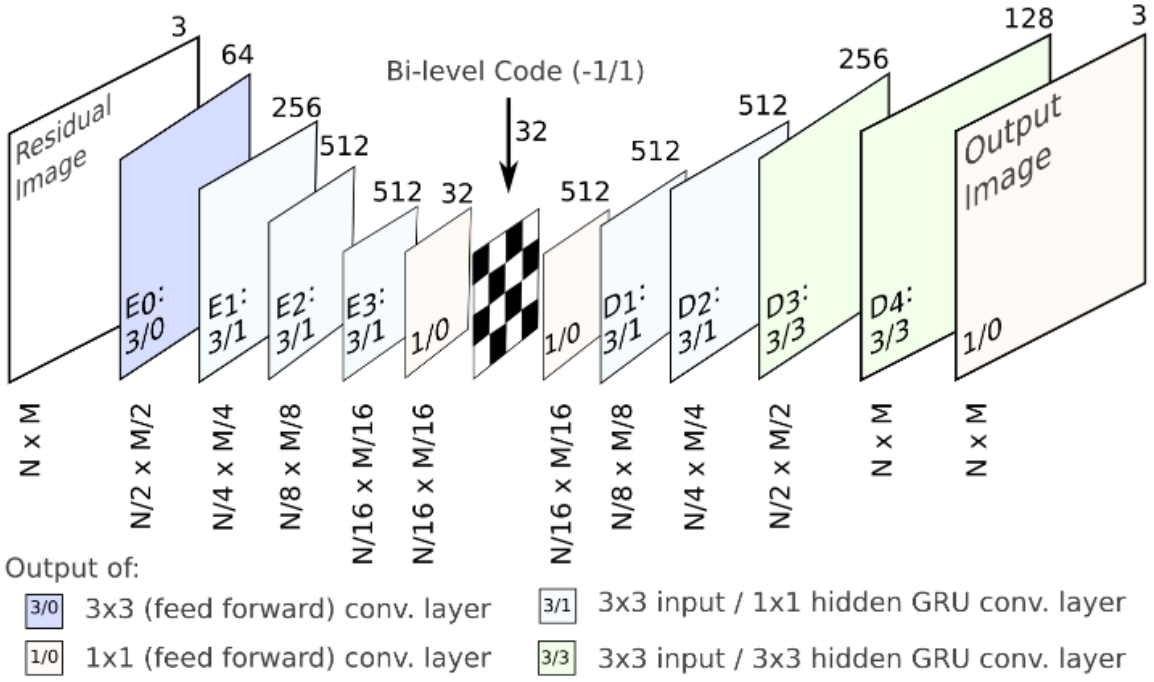

Рис. 2.14 - Схема рекурентної моделі нейронної мережі[22]

Оскільки значення вихідної реконструкції від бітових стеків залежить від положення вихідного пікселя на один бітовий стек (в кожному просторовому вимірі), то нам слід обговорити тільки максимальну просторову підтримку:

- $max(S_B(F_t)) = 6t + [5.5]$
- $S_I(F_t) = 16S_B(F_t) + 15$

де $S_B(F_t) \times S_B(F_t)$ і $S_I(F_t) \times S_I(F_t)$ є просторовою підтримкою реконструкції на бітових стеках і на пікселях вихідного зображення, відповідно.

На першій ітерації даних мереж стиснення прихованого стану кожного рівня GRU[23] ініціалізується нулем. У проведених експериментах можна спостерігати сильне візуальне поліпшення якості зображення протягом перших кількох ітерацій. Отже основна гіпотеза полягає в тому, що відсутність гарної ініціалізації прихованого стану погіршує основні ранні характеристики швидкості передачі даних. Оскільки архітектури кодера і декодера складають кілька рівнів GRU послідовно, потрібно кілька ітерацій, щоб поліпшення

прихованого стану з першого рівня GRU стало помітним в бінарізаторі (для кодувальника) або при реконструкції (для декодера).

Даний підхід до вирішення цієї проблеми полягає в тому, щоб згенерувати краще початкове значення прихованого стану для кожного шару за допомогою техніки, так званого праймінгу прихованого стану. Праймінг в прихованому стані, або «k-праймінг», збільшує поточну глибину першої ітерації мереж кодера і декодера окремо на додаткові кроки. Щоб уникнути використання додаткової смуги пропускання, ми виконуємо ці додаткові кроки окремо, не додаючи додаткові біти, що створюються кодувальником, до фактичного потоку бітів. Для кодувальника це означає багаторазову обробку вихідного зображення, відкидаючи згенеровані біти, але зберігаючи зміни прихованого стану в повторюваних одиницях кодувальника. Для декодера це означає, що береться перший дійсний набір переданих бітів і генерується декодувати зображення кілька разів, але зберігається тільки остаточна реконструкція зображення (і зміни прихованих станів декодера).

Праймінг також можна виконувати між ітераціями. Даний процес називається ще дифузією. Експериментально дифузія показала кращі результати, але за рахунок часу виконання і часу навчання. Крім досягнення кращого уявлення прихованого стану для даних мереж, попередня обробка та розповсюдження також збільшують просторову протяжність прихованих станів в декодері[24] , Де останні два рівня прихованих ядер мають розмір 3 × 3, і в більш пізніх ітераціях кодера, коли збільшена підтримка декодера поширюється на збільшену підтримку кодера.

2.2.2 Карта важливості для просторового розподілу бітів

Гладкі області зображення легше стиснути, ніж області з виступаючими об'єктами або багатими на деталі текстурами. Таким чином, менша кількість бітів повинна бути виділена гладким областям, в той час як більше бітів повинно бути виділено областям з високим інформаційним змістом. Більш того, коли повна довжина коду для зображення обмежена, такі схеми розподілу також можуть використовуватися для управління ступенем компресії. Зважена за

змістом карта важливості вводиться для розподілу бітів і управління ступенем стиснення.

Це карта характеристик тільки з одним каналом, і її розмір повинен бути таким же, як і у вихідних даних кодувальника. Значення карти важливості знаходиться в діапазоні (0,1). Мережа карти важливості розгортається для вивчення карти важливості з вхідного зображення x. Вона приймає проміжні карти характеристик $f(x)$ з останнього залишкового блоку кодера в якості вхідних даних і використовує мережу з трьох згорткових шарів для створення карти важливості $p = P(x)$.

Нагадаємо, що, оскільки E і D є CNN, z_{is} є 3D картою ознак мережі. Наприклад, якщо E має згортковий шар з трьома кроками 2 і вузьке місце має K каналів, отже розміри Z будуть $\frac{W}{8} \times \frac{H}{8} \times K$. Наслідком цього формулювання[25] є те, що ми використовуємо однакову кількість символів в z для кожного просторового положення вхідного зображення x. Однак відомо, що на практиці інформаційний зміст сильно розрізняється по просторовим точкам (наприклад, однорідна область блакитного неба в порівнянні з дрібно зернистою структурою листя дерева). В принципі, це може бути враховано автоматично при виборі компромісу між ентропією і спотворенням, коли мережа навчиться виводити більш передбачувані[26] (тобто з низькою ентропією) символи для областей з низькою інформацією, звільняючи місце для використання символи ентропії для більш складних регіонів.

Точніше, формулювання в уже допускає змінний розподіл бітів для різних просторових регіонів через контекстну модель P. Однак це, можливо, вимагає досить складної (і, отже, дорогої в обчислювальному відношенні) контекстної моделі замість використання карти важливості, щоб допомогти CNN приділяти увагу різним регіонам зображення із різною кількістю біт. Хоча для цієї мети використовується окрема мережа, ми розглядаємо спрощену настройку. Ми беремо останній шар кодувальника E і додаємо другий одноканальний вихід

$y \in R^{\frac{W}{8}} \times \frac{H}{8} \times 1$. Ми розширюємо цей єдиний канал в[27] маску $m \in R^{\frac{W}{8}} \times \frac{H}{8} \times K$ тієї ж розмірності, що і Z, в такий спосіб. Це змінює $max(S_B(F_t))$ на

$$max(S_B(F_t)) = \left[1.5k_d + 5.5\right]t + \left[1.5k_p + 5.5\right], \text{ де } k_p = k_d, \text{ при } k_d > 0$$

де $y_{i,j}$ позначає значення y в просторовому місцезнаходження (i, j). Значення переходу для $k \leq y_{i,j} \leq k + 1$ таке, що маска плавно переходить від 0 до 1 для не цілих значень y. Ми модифікуємо маску z точковим множенням з бінаризацією m, тобто $z \leftarrow z \odot [m]$. Оскільки оператор округлення [•] не диференціюється. За допомогою цієї модифікації ми просто трохи змінили архітектуру E, щоб вона могла легко «обнулити» частини стовпців $z_{i,j}$: $of\ z$. Як було запропоновано, отримана таким чином структура в z представляє альтернативну стратегію кодування: замість кодування без втрат всього обсягу символу z ми могли б спочатку (окремо) кодувати маску, а потім для кожного стовпця $z_{i,j}$ кодувати тільки перші $[m_{i,j}] + 1$ символів, так як інші є константою Q (0), яку ми називаємо нульовим символом. Робота використовує двійкові символи (тобто C = {0,1}) і передбачає незалежність між символами і однаковими попередніми символами під час навчання, тобто вартість кожного біта для кодування.

Таким чином, карта важливості є їх основним інструментом для управління бітрейтом, оскільки таким чином вони уникають кодування всіх бітів в поданні. Навпаки, ми дотримуємося формулювання, в якій залежності між символами моделюються під час навчання. Потім ми використовуємо карту важливості як архітектурного обмеження і використовуємо запропоновану ними стратегію кодування для отримання альтернативної оцінки ентропії $H(z)$ наступним чином. Ми помічаємо, що ми можемо відновити Dme з z, підрахувавши кількість послідовних нульових символів в кінці кожного стовпчика $z_{i,j}$. Отже $[m]$, є функцією[28] замаскованого z, тобто $[m] = g(z)$для

g, що відновлює $[m]$ як описано, що означає, що для умовної ентропії $H([m]|z) = 0$. Тепер ми маємо

$$H\left(\widehat{z}\right) = H\left([m]|\,\widehat{z}\right) + H\left(\widehat{z}\right) = H\left(\widehat{z},\,[m]\right) = H(\,\widehat{z}\,|\,[m]) + H([m])$$

Якщо ми розглядаємо ентропію маски H (dme) як постійну під час оптимізації авто-кодувальника, ми можемо потім побічно мінімізувати $H(\widehat{z})$ через $H(\,\widehat{z}\,|\,[m])$. Щоб оцінити $H(\,\widehat{z}\,|\,[m])$, ми використовуємо ту ж факторизацію p, але оскільки маска $[m]$ відома, ми маємо $p\,(\widehat{z_i} = c_0) = 1$ детермінований для тривимірних розташування i в z, де маска дорівнює нулю. Журнали відповідних термінів потім оцінюються як 0. Решта членів ми можемо змоделювати з тієї ж контекстної моделлю $P_{i,l}\,(z)$, що призводить до

$$H(\,\widehat{z}\,|\,[m]) \approx E_{\widehat{z}\sim p(\widehat{z})}\left[\sum_{i=1}^{m} - \left[m_i\right] log P_{i,l\left(\widehat{z_i}\right)}\right]$$

2.2.3 Просторово адаптивний бітрейт

За своєю конструкцією наші рекурентні моделі генерують представлення зображень з різною швидкістю передачі відповідно до кількості ітерацій, але ці швидкості передачі постійні для кожного зображення[22]. Це означає, що локальна (номінальна) бітова швидкість фіксується незалежно від складності основного вмісту зображення, що неефективно з точки зору кількісного і сприйманого якості (наприклад, розгляньте кількість бітів, необхідних для точного кодування чіткого зображення. На практиці ентропійний кодер вводить деяку просторову адаптивність, засновану на складності та передбачуваності двійкових кодів, але наша процедура навчання не заохочує кодувальник безпосередньо генерувати коди з низькою ентропією. Замість цього функція втрат тільки підштовхує мережу до максимального підвищення якості реконструкції за фрагментами зображення. Щоб забезпечити максимальну якість всього зображення при цільовій (середній) швидкості передачі даних , ми вводимо процес постпроцесингу просторово-адаптивної швидкості передачі

даних для динамічного налаштування локальної швидкості передачі даних відповідно до цільової якості реконструкції.

При заданій якості кожного фрагменту зображення призначається стільки бітів, скільки необхідно для досягнення або перевищення цільової якості до максимуму, підтримуваного моделлю. Оскільки описаний декодер використовує чотири шари 2 × 2 глибини в простір, кожен бітовий стек відповідає шару зображення розміром 16 × 16. Модель використовує до 16 ітерацій, кожна з яких додає 32 біта до кожного стека, тому алгоритм розподілу виконує призначення з кроком 32. Ми обчислюємо якість кожного шару як максимум середнього значення помилки L1 для чотирьох суб-шарів розміром 8 × 8, тому що усереднений повний суб шар розміром 16 × 16 привів до видимих артефактів, які охоплюють як прості, так і візуально складні ділянки зображення.

Нарешті, застосовується евристика, згідно з якою всі шари повинні використовувати від 50% до 120% цільової[29] швидкості передачі даних (з округленням до найближчої ітерації), щоб уникнути візуальних артефактів. Ми очікуємо, що використання більш точної метрики сприйняття зробить цю евристику непотрібною. Дана архітектура декодера вимагає, щоб були присутні всі біти, тому ми заповнюємо відсутні біти фіксованим значенням. Хоча мережа була навчена відображенням двійкових значень в ± 1, було виявлено, що використання значення нуля призвело до кращої якості відновлення. Вважається, що нуль працює добре в першу чергу тому, що згорткові шари використовують заповнення нулями, що підштовхує мережу до розуміння того, що нульові біти неінформативні. Нуль також знаходиться на півдорозі між стандартними значеннями бітів, які можна інтерпретувати як значення з найменшим зміщенням, і це поглинає елемент для множення, який мінімізує індукований відгук при пакунку.

SABR(spatially adaptive bit rate) вимагає невеликого додавання до потоку бітів, згенеровані нашою моделлю, щоб декодер знав, скільки бітів використовується в кожному місці. Ця карта важливості стискається без втрат за допомогою gzip і додається в потік бітів. Щоб забезпечити чесне порівняння,

загальний розмір цих метаданих включений в усі обчислення швидкості передачі даних.

ВИСНОВКИ ДО РОЗДІЛУ 2

В даному розділі було розглянуто поняття нейронної мережі, її основні типи та компоненти. Розглянуто структуру компонентів та випадки використання різних моделей нейронних мереж та їх недоліки. Серед яких:

- Рішення не є оптимальним, якщо все ймовірності не є негативними ступенями 2. Це означає, що в більшості випадків існує розрив між середнім числом бітів і ентропією.
- Незважаючи на наявність методів для досить швидкого підрахунку частоти кожного символу, відновлення всього дерева для кожного символу може бути дуже повільним. Зазвичай це відбувається, коли алфавіт великий і розподіли ймовірностей швидко змінюються з кожним символом.

Для вирішення проблеми компресії даних цифрового зображення розглянуто:

- Використання рекурентної нейронної мережі с довго-короткотривалою пам’яттю для кодувальника/декодувальника задля забезпечення коректної реконструкції стиснутого зображення.
- Використання карти важливості для підвищення точності нейронної мережі у процесі генерування нового зображення на основі створеного минулою ітерацією.

Також, важливо зазначити, що були розглянуті структура та основні проблеми обраних нами для реалізації задачі методів.

РОЗДІЛ 3

РОЗРОБКА МЕТОДУ СТИСНЕННЯ ЗОБРАЖЕННЯ З ВИКОРИСТАННЯМ НЕЙРОННОЇ МЕРЕЖІ

3.1 Розробка методу стиснення зображення

В даному розділі описується розробка нейронної мережі що виконує операцію компресії цифрового зображення.

Після розгляду кількох найбільш поширених та нових методів стиснення зображень, в тому числі на основі нейронних мереж, було прийнято рішення дослідити метод глибокої компресії на основі квантування змінних у каналах зображення з використанням нейронної мережі.

Рішення було прийнято з огляду на сучасність методу та можливість отримання кращих якісних результатів у процесі власного дослідження. Основною мовою програмування є Python, основним фреймворком для реалізації нейронної мережі є Tensorflow в комбінації із бібліотекою Keras[30] та PyTorch[31]. Для тренування нейронної мережі використовується декілька відкритих наборів даних від Div2k[32,34,35,36]. Всі діаграми створені з допомогою Draw.io. Основними провайдерами віртуальних машин є Microsoft Azure та Google Colab.

3.2 Структура мережі

За основу нашої роботи ми взяли дослідження[37] з стиснення зображення з допомогою квантування по каналам. Отже створена розроблема система буде створена на основі наступних артефактів її структури на Рис. 3.1.

- Ко/Декодувальник

Наш кодер залишкового уваги каналу складається з трьох частин: голови, тіла і хвоста. Головний модуль містить один згортковий шар, який перетворює вихідне зображення в карту характеристик $X(0)$ з каналами $Y(0)$. Корпус кодувальника показаний на ілюстрації зліва. Все тіло включає чотири стадії. На кожному етапі вихідна карта функцій є тільки половину рішення (h, w) вхідної карти функцій. Позначимо карту вхідних характеристик на

етапі t як $X(t) \in R^{C_t \times H \times W}$. Керуючись методом задання надвисокої роздільної здатності, ми використовуємо зворотній метод реорганізації пікселів (PixelShuffle) - змінює форму тензора з

$$(*,\ C \times r^2,\ H,\ W) \text{ до } (*,\ C,\ H \times r,\ W \times r)$$

для реалізації операції понижувальної дискретизації. Яку можна виразити наступною формулою:

$$IPS(X^{(t)})_{c(di+j),h,w} = X^{(t)}_{c,\ dh+i,\ dw+j},\ 1 \le i,\ j \le d,$$

де d - коефіцієнт понижувальної дискретизації. Це періодичний оператор перестановки, який перетворює елементи тензора $C_t \times H \times W$ в тензор формою $d^2 C_t \times \frac{1}{d} H \times \frac{1}{d} W$.

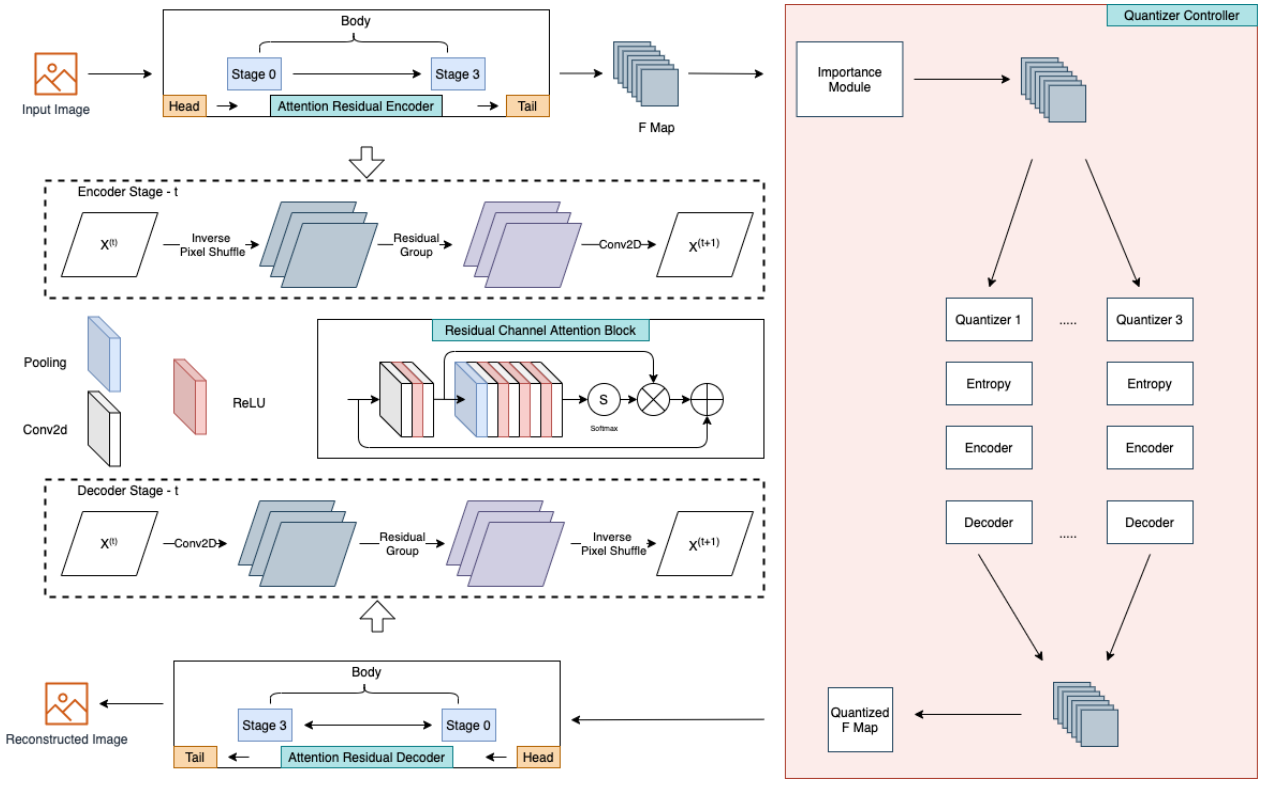

Рис. 3.1 - Модель методу компресії

Слід зауважити, що цей оператор зберігає всю інформацію про вхідні дані, тому що кількість елементів не змінюється.

Також відомо, що зворотний реорганізації пікселів може поліпшити стабільність навчання і знизити витрати пам'яті в порівнянні з згорткою із

понижаючою дискретизацією Попередні методи стиснення зображень створені на основі CNN однаково обробляють поканальні функції, що не є гнучким для реальних випадків.

Щоб мережа була зосереджена на найбільш інформативних функції і використовувала взаємозалежності між каналами функцій. Ми відправляємо карту характеристик в модуль залишкової групи, показаний у лівій частині ілюстрації. Залишкова група складається з B блоків уваги залишкового каналу, які використовуються для вилучення взаємозалежностей між каналами ознак і виділення карти ознак.

Залишкова група не змінює кількість каналів. Нарешті, ми додаємо згортковий шар, щоб змінити кількість каналів з C_t на $C_t + 1$ для наступного етапу. Таким чином, вихід кроку t:

$$X^{(t+1)} \in R^{C_{t+1} \times \frac{1}{d}H \times \frac{1}{d}W}$$

який також є входом для наступного етапу.

Після чотирьох етапів обробки в тілі, згортковий шар, створений у хвостовій частині, генерує стисле (приховане) уявлення Z з каналами C, де C можна змінювати вручну для різних BPP. Точно так же архітектура декодера є просто інверсною версію кодера. Як показано, ми замінюємо зворотний PixelShuffle на PixelShuffle для операції підвищення дискретизації.

Особливістю даного компоненту системи є збільшений розмір блоку що відповідає за залишкове значення уваги каналу. Основне дослідження використовувало стандартний підхід з роботи Йоханеса Балля[38], що описував використання блоку складеного з наступних шарів:

1. Con2D
2. ReLU
3. Conv2D

які замикаються сигмоїдною функцією активації. Оскільки немає на даний момент конкретних рекомендації щодо використання сигмоїдної функції у

автоматичних кодувальниках, як у випадку з класифікаторами де все залежить від типу зображень які класифікатор буде обробляти, було вирішено використати функцію софтмакс. Блок уваги залишкового каналу був змінений до наступного вигляду:

1. Con2D
2. ReLU
3. Con2D
4. ReLU
5. Conv2D
6. ReLU
7. Conv2D

де перші 4 шари відповідаю за downscale а останні 4 - upscale.

- Квантувальник

Для квантувальника ми пропонуємо наступний метод квантування, заснований на GMM. А сами, ми моделюємо попереднє розподілення $p\ (Z)$ як суміш гаусових розподілень:

$$p(Z) = \prod_{i} \sum_{q=1}^{Q} \pi_q N(z_i|\mu_q, \sigma^2_q)$$

де π_q, μ_q і σ_q - параметри навчання гаусової суміші, а Q - рівень квантування. Ми отримуємо результат прямого квантування, встановлюючи його на середнє значення, яке вимагає максимальної відповідальності:

$$\hat{z}_i \leftarrow arg_{\mu_j} max \frac{\pi_j N(z_i|\mu_j, \sigma^2_j)}{\sum_{q}^{Q} \pi_q N(z_i|\mu_q, \sigma^2_q)}$$

Очевидно, даний вираз не є диференційованим. Ми виконуємо операцію релаксації $\hat{z}_i$ до $\bar{z}_i$, щоб обчислити його градієнти під час зворотного проходу повз:

$$\overline{z_i} = \sum_{j=1}^{Q} \frac{\pi_j N(z_i|\mu_j,\sigma_j^2)}{\sum_q^Q \pi_q N(z_i|\mu_q,\sigma_q^2)} \mu_j$$

На відміну від звичайного GMM, який оптимізує π_q, μ_q і σ_q за допомогою EM(expectation-maximization)- алгоритму максимізації очікування, ми вивчаємо параметри суміші, мінімізуючи функцію втрат з негативним правдоподібністю через зворотне поширення мережі. Позначимо функцію втрат апріорного розподілу GMM-квантувальника як:

$$L_{GMM} = -log(p(Z)) = -\sum_i log \sum_{q=1}^{Q} \pi_q N(z_i|\eta_q, \sigma_q^2)$$

Тут ми хотіли б провести порівняння між квантувальником GMM і м'яким квантувальником. М'який квантувальник можна розглядати як диференційовану версію алгоритму K-середніх. Якщо параметри суміші задовольняють:

$$\pi_1 = \pi_2 = \cdots = \pi_Q = 1/Q \text{ і } \sigma_1 = \sigma_2 = \cdots = \sigma_{\text{й}} = \sqrt{2}/2$$

квантувальник GMM буде вироджений в м'який квантувальник, що означає, що квантувальник GMM має більш потужне уявлення і є більш узагальненою моделлю.

- Контролер квантизації

Кожен канал квантизованої карти характеристик може по-різному впливати на остаточні результати реконструкції. Щоб виділити відповідні бітрейти для різних каналів, ми пропонуємо модель контролера змінного квантування. Ілюстрація контролера змінного квантування показана в правій частині. У контролері змінного квантування є два ключові компоненти: модуль важливості каналу і модуль поділу-злиття.

- Модуль важливості каналу

Входом модуля важливості каналу є Z, який є виходом кодувальника. Позначимо номер каналу Z як C ($C = 8$). Ми очікуємо, що модуль важливості каналу згенерує вектор важливості каналу $w \in R^{C}_{+}$. Кожен елемент wc являє собою конструктивну важливість каналу c. Тут ми розробляємо три типи модуля важливості каналу:

- Стискання і збудження блоків

Ми використовуємо середній пул і два згортальних шару для роботи з Z і отримання матриці $M \times C$,де M- розмір міні-пакета. Ми генеруємо якого навчають вектор важливості каналу, використовуючи операцію середнього над матрицею, зменшуючи перший вимір.

- Реконструкція на основі помилок

Ми виконуємо три кроки для його реалізації. По-перше, ми створюємо набір даних для перевірки, випадковим чином вибираючи N зображень з набору навчальних даних. По-друге, ми обрізаємо c-й канал карти функцій n-го зображення:

$$Z_{n,c}: Z_n(c,:,:) = 0$$

Нарешті, ми представляємо w_c, обчислюючи середню помилку реконструкції MS-SSIM(multiscale structure similarity index) для кожного каналу набору даних перевірки:

$$w_c = \frac{1}{N}\sum_{n=1}^{N} d_{MS-SSIM}(I_n, Dec(Qua(Z_{n,c})))$$

де I_n - n-е зображення набору даних перевірки, Dec і Qua - уявлення декодера і квантувача відповідно.

- Predefined

Ми безпосередньо задаємо вектор важливості каналу $w_c = c$, який фіксується в процесі навчання і оцінки.

- Модуль Поділу-Об'єднання

На початку модуля поділу-об'єднання ми сортуємо функцію $map\ Z_{in}$ в порядку зростання відповідно до вектору важливості каналу w. Оскільки нова карта функцій добре організована, ми розділили її на групи G[39]. Частини G карти ознак квантуються і кодуються з використанням різних рівнів квантування в різних групах. Після операції поділу[40] C-канали поділяються на G-групи. Позначимо вектор відносин G-груп as r, який задовольняє:

$$\sum_{g}^{G} (r_g) = 1, \text{ i } \forall g, r_g > 1$$

Тут ми використовуємо праву частину рис. 2, щоб пояснити його механізм. Припустимо, що параметри:

$$C = 8, G = 3, \text{ i } r = [25\%, 50\%, 25\%]^T$$

Канали 1 і 2 будуть призначені групі 1 для квантування і кодування, канали 3, 4, 5 і 6 будуть призначені групі 2, а канали 7 і 8 будуть призначені групі 3.

З іншого боку, оскільки значення каналів у каналів 1 і 2 менше, ніж у інших, ми використовуємо менший рівень квантування q_1 для квантування і кодування. Точно так само ми застосовуємо більш високий рівень квантування q_3 для квантування і кодування каналів 7 і 8. На останньому етапі ми об'єднуємо групи G і міняємо порядок вимірювання каналу, щоб побудувати остаточний стислий результат.

3.3 Тренування мережі

3.3.1 Датасет

Для навчання ми використовували набори даних Div2k, а саме бікубічний та high-res Div2k. які містять всього близько 1800 зображень. Дотримуючись багатьох методів глибокого стиснення зображень, ми оцінили наші моделі на наборі даних Kodak[41] з метриками PSNR(peak signal to noise ratio), SSIM(structural similarity index), MS-SSIM(multiscale structural similarity index), BPP(bits per pixel) для стиснення зображень з втратами. Важливо зазначити що наш тренувальний датасет складається с як тренувальної так і валідаційної

частини Div2k дата сетів, оскільки задля валідації ми приміняємо інший набір даних то використання валідаційної частини надає можливість легко збільшити кількість зображень із такими самими характеристиками.

3.3.2 Оригінальні гіпер параметри

У оригінальному дослідженні було використано наступні основні гіпер параметри для тренування створеної моделі нейронної мережі на :

Назва	Значення
Робітник	16
Рівні квантувальника	[3,5,7]
Розмір вибірки	32
Швидкість навчання кодувальника	0.0001
Швидкість навчання де-кодувальника	0.0001
Швидкість навчання ентропії	0.0001
Швидкість навчання квантувальника	0.00005
Віха	[200, 300, 350]
Кількість епох	400
Гамма	0.2

Табл. 3.1- Загальні параметри нейронної мережі

Зображений в Табл. 3.1 параметр **Workers** відповідає за кількість паралельно запущених процесів тренування та дорівнює кількості фактичний ядер процессора, **QUA Levels** - значення кожного рівня кватнизації, **Leaning Rate** [Encoder | Decoder | Entropy | Quantizer] - крок навчання для кодувальника, декодувальника, ентропії та квантизатора відповідно, **Gamma** - коефіцієнт зменшення кроку, **Milestone** - номер епохи на якій буде зменшено розмір кроку. **Epoch Size** - кількість епох тренування. Табл. 3.2 відображає основні параметри моделі нейронної мережі що тренується за SE підходом. За SE методом ми генеруємо вектор важливості каналу w, використовуючи операцію знаходження середнього значення над матрицею, зменшуючи перший вимір (M).

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[32, 64, 128, 198]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[192, 128, 64, 32]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)]
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]
Ознака Ентропії	[192, 192, 192]

Табл. 3.2 Набір SE гіпер параметрів

.

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[32, 64, 128, 198]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[192, 128, 64, 32]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)]
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]
Ознака Ентропії	[192, 192, 192]

Табл. 3.3 Набір RE гіпер параметрів

Табл. 3.3 відображає основні параметри моделі нейронної мережі що тренується за RE[42] підходом.

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[32, 64, 128, 198]
Значення блоку де-кодувальника	[6, 6, 6, 6]

Значення ознак де-кодувальника	[192, 128, 64, 32]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)]
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]
Ознака Ентропії	[192, 192, 192]

Табл. 3.4 Набір Predefine гіпер параметрів

Табл. 3.4 відображає основні параметри моделі нейронної мережі що тренується за Predefine підходом.

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[32, 64, 128, 198]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[192, 128, 64, 32]
σ Квантувальника	[(x5 0.7072)]
π квантувальника	[x5 0.2]
Частина Ентропії	[1]
Ознака Ентропії	[192]

Табл. 3.5 Набір Baseline гіпер параметрів

Табл. 3.5 відображає основні параметри моделі нейронної мережі що тренується за Predefine підходом.

3.3.3 Наші гіпер параметри

Обрані нами гіпер параметри зображені на Табл. 3.6 - Табл. 3.10 нижче.

Назва	Значення
Робітник	6
Рівні квантувальника	[3,5,7]
Розмір вибірки	4
Швидкість навчання кодувальника	0.0001

Швидкість навчання де-кодувальника	0.0001
Швидкість навчання ентропії	0.0001
Швидкість навчання квантувальника	0.00005
Віха	[30, 55, 80]
Кількість епох	100
Гамма	0.2

Табл. 3.6 Загальні параметри нейронної мережі

Набір SE гіпер параметрів	
Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[64, 128, 198, 256]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[256, 192, 128, 64]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]
Ознака Ентропії	[256, 256, 256]

Табл. 3.7 Загальні параметри нейронної мережі

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[64, 128, 198, 256]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[256, 192, 128, 64]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]

Ознака Ентропії	[256, 256, 256]

Табл. 3.8 Набір RE гіпер параметрів

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[64, 128, 198, 256]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[256, 192, 128, 64]
σ Квантувальника	[(x3 0.7072),(x5 0.7072),(x7 0.7072)
π квантувальника	[(x3 0.3333), (x5 0.2), (x7 0.1429)]
Частина Ентропії	[0.25, 0.5, 0.25]
Ознака Ентропії	[256, 256, 256]

Табл. 3.9 Набір Predefine гіпер параметрів

Назва	Значення
Значення блоку кодувальника	[6, 6, 6, 6]
Значення ознак кодувальника	[64, 128, 198, 256]
Значення блоку де-кодувальника	[6, 6, 6, 6]
Значення ознак де-кодувальника	[256, 192, 128, 64]
σ Квантувальника	[(x5 0.7072)]
π квантувальника	[x5 0.2]
Частина Ентропії	[1]
Ознака Ентропії	[256]

Табл. 3.10 Набір Baseline гіпер параметрів

3.3.4 Апаратне забезпечення

Для тренування ми використовували 2 основних провайдери віртуальних машин:

- Microsoft Azure
- Google Colab

Такий вибір був зроблений через неможливість використання Google Colab для довготривалих процесів навчання, адже час життя сесії максимально може дорівнювати 12 годинам що значно менше ніж наш процес тренування. За основу було обрано віртуальну машину NC6 на Microsoft Azure із наступною конфігурацією:

- CPU - 6 ядер
- RAM - 56 ГБ
- GPU - 1 Nvidia K80 з 12 ГБ пам’яті
- Hard Drive - 375 ГБ
- ОС - Azure OS що базується на Ubuntu
- Python - 3, автоматично йде в комплекті із Azure OS
- Ціна - 1.16 $ / год.

Розмір пакета зменшено через обмеження пам'яті на GPU оскільки загальне використання пам'яті графічного процесора потребує більше 24 ГБ відеопам’яті. Навчання нейронної мережі може бути значно прискорене використання декількох графічних процесорів у комбінації із багато ядерним процесором, найкращим вибором буде процесори типу Ryzen. У майбутніх дослідженнях ми також протестуємо процесори TPU, що має значно прискорити процес навчання.

ВИСНОВКИ ДО РОЗДІЛУ 3

У даному розділі були розглянуті використання рекурентної нейронної мережі у комбінації з генералізованим квантувальником та різними типами контролерів квантизації. Детально описано теоретичну складову кожного контроллера та їх різницю.

Було створено програмну реалізацію запропонованої моделі. Модель була успішно натренована на відкритому наборі даних Div2k що складається з 1800 зображень високої якості. Був досягнутий ефект компресії зображення при незначних вратах якості зображення.

У наступному розділі ми розглянемо метрики розробленої моделі більш детально та порівняємо її із вже існуючими реалізаціями даної проблеми.

РОЗДІЛ 4

ПОРІВНЯННЯ РЕЗУЛЬТАТІВ ІЗ ІСНУЮЧИМИ ТЕХНОЛОГІЯМИ

Даний розділ присвячений розгляду отриманих під час тренування результатів та розгляду експериментальних результатів що були отримані під час пошуку оптимального вирішення поставленою магістерською дисертацією проблеми.

Рис 4.1 - Рис. 4.3 зображують навантаження системи та поточне використання відеопам’яті під час процесу тренування.

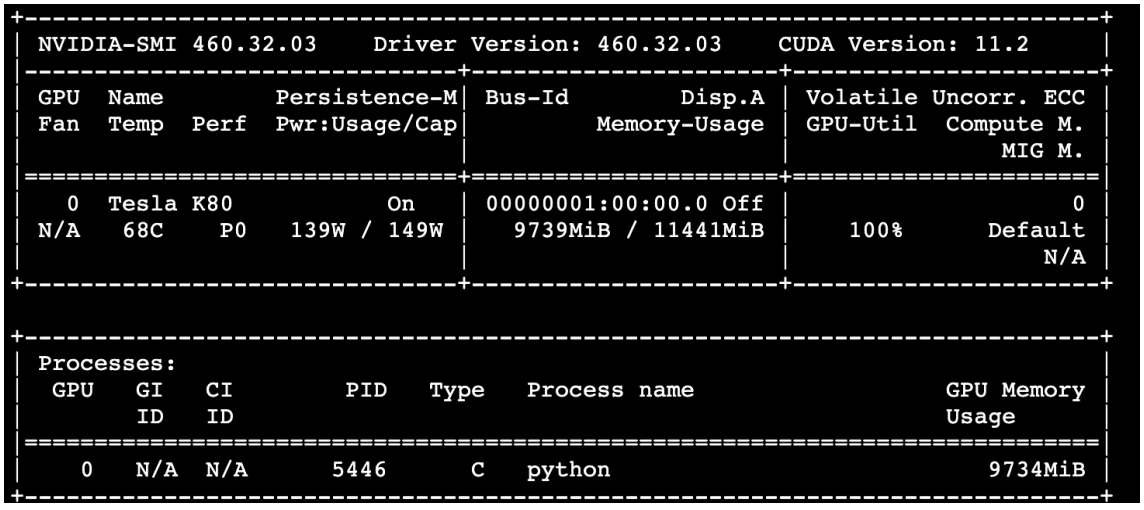

Рис. 4.1 - Завантаження відеокарти на 1 хвилині тренування

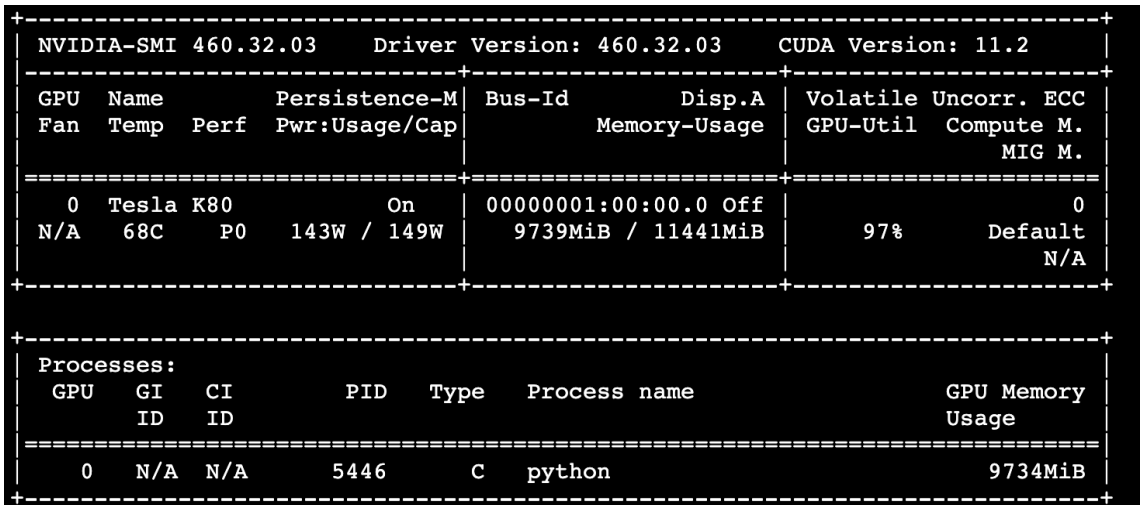

Рис. 4.2 - Завантаження відеокарти на 10 хвилині тренування

Як ми можемо спостерігати під час використання нашої тренувальної конфігурації розробленого методу глибокої компресії зображення, використання пам’яті є статичним увесь час та займає 9734 МБ із 11441 МБ загальної відео пам’яті, що дорівнює 85% від загального об’єму.

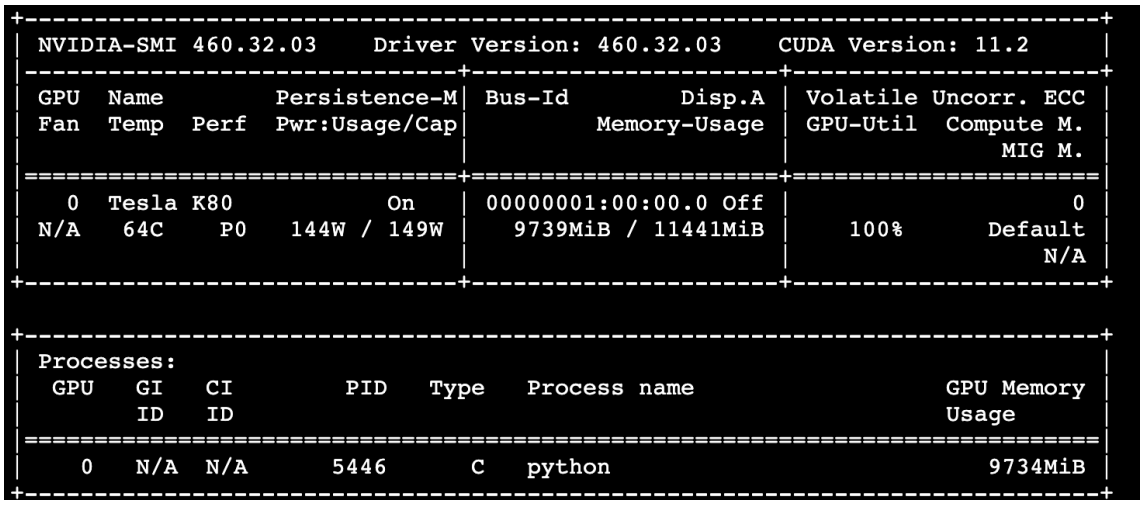

Рис. 4.3 - Завантаження відеокарти на 60 хвилині тренування

Дані результати дозволяють нам зробити висновок що нейронна мережа досить ефективно використовує пам’ять відео процесора хоча і залишається простір для оптимізації. Також цікаво зазначити ріст напруги відносно часу використання при статичному значенні температури чіпу. Утилізація відео процесору протягом часу тренування є максимальною з $\Delta \sim 3$ %, що дає нам зробити висновок про оптимальну утилізацію ресурсів відеокарти.

Нейронна мережа пройшла тренування у 100 епох. Загальний час тренування займає приблизно 40-50 годин, довжина однієї ітерації займає в середньому 25-30 хвилин що підтверджують дані зображені на Рис. 4.8. Також добре видно що наявні стрибки часу тренування на різних ітераціях де час займає більше ніж одну 60 хвилин, проте оскільки таких спонтанних стрибків не багато і тенденції немає можна списати це на похибку. Проаналізуємо результати оброблених зображень. Як вже було зазначено для валідації був використаний відкритий датасет Kodak що використовується у більшості випадків для тестування моделей нейронних мереж що пов’язані з підвищенням якості обробленого зображення, так званим апскейлінгом, і у нейронних мережах з глибокої компресії зображень. Також слід зауважити що під час процесу тренування кожна ітерація виконувала по три експерименти відповідно та її результатом є середнє (mean) значення що було обраховане стандартними бібліотеками для python, метрики для перехресної валідації зображення також

обраховані з використанням бібліотек python таких самих як у основному дослідженні за для коректного кількісного порівняння.

Рис. 4.4 - Оригінальне зображення із датасету Kodak
MS-SSIM / PSNR / BPP

Рис. 4.5 - Реконструйоване зображення оригінальною нейронною мережею
0.935 / 24.253 / 0.261

Через неможливість відтворити таку значну кількість епох на власному обладнанні ми натренували оригінальну нейронну мережу на нашому датасеті, описаного в 3 розділі,з оригінальними параметрами задля якісного порівняння даних метрик. Розглянемо детальніше отримані метрики:

- MS-SSIM - відповідає за структурну подібність зображення, може набирати значення від 0 до 100, фактично є процентом на який зображення подібне до оригінального. Чим вище тим краще
- BPP (Bits Per Pixel) - відповідає за кількість бітів у відному реконструйованому зображенні, немає зв’язку із оригінальним зображенням. Фактично чим менше значення тим краща компресія

- PSNR (Peak Signal To Noise Ratio)- відповідає за значення пікового сигналу зображення до шуму. Чим вище значення пікового сигналу тим краща якість стиснутого і/або реконструйованого зображення відносно протестованого оригіналу

Рис. 4.6 - Реконструйоване зображення 1 нашою нейронною мережею

0.963 / 25.647 / 0.238

Рис. 4.7 - Реконструйоване зображення 2 нашою нейронною мережею

0.966 / 25.632 / 0.240

Як ми можемо побачити із Рис. 4.5 оригінальний метод компресії показав досить високі значення навіть на валідаційному дата сеті. Незважаючи на те що мережа тренована на нашому наборі даних, описаному у 3 розділі, із зменшеною кількістю зображень. Проте відносно оригіналу на Рис. 4.5 яскраво видно втрату кольору при досягненні коректної контрастності. Також має місце значна пікселізація відносно оригіналу на Рис. 4.4 та надлишкові артефакти особливо яскраво даний феномен видно на частині зображення із Рис. 4.8 видно що деталізація вища на зображенні що реконструйоване нашою нейронною

мережею адже лише у нас на Рис. 4.7.видно останню цифру 5 із всього номеру на парусі. Дана особливість забезпечена вищим значенням метрики MS-SSIM.

Рис. 4.8 - Порівняння артефактів на реконструйованому зображенні. Зліва направо - Оригінальне зображення, реконструйоване оригінальне, реконструйоване наше.

Розглянемо детальніше отримані метрики процесу тренування на Рис. 4.9. Як ми можемо побачити позитивна динаміка спостерігається на метриках MS-SSIM та PSNR, тенденція показує нам що значення стабільно росте вверх дозволяючи нам значно збільшити якість результату нейронної мережі зменшивши кількісно об’єм епох у 4 рази, потенційно навіть незначне збільшення датасету та підйом кількості епох до 200 може дозволити нам перевищити навіть значення досягнуте у оригінальному дослідженні.

Проте незважаючи на позитивну динаміку з боку даних двох метрик видно що кількість бітів на піксель зростає із кожною новою епохою сповільнюючись разом із швидкістю росту перших двох метрик. Дане спостереження дає нам висунути гіпотезу що потенційно коефіцієнт компресії падає разом із підвищенням якості зображення. Порівняємо фактичні розміри зображення:

- Оригінальне зображення - 583 Kb
- Реконструйоване зображення оригінальною мережею - 441 Kb, 75.6 %
- Реконструйоване зображення нашою мережею - 430 Kb, 73.7 %, що нижче 1.9 % на Рис. 4.6

Загалом результат компресії отриманий із використанням нейронної мережі значно відстає від класичних методів. Якщо взяти до уваги такі методи як JPEG, TIFF то вони можуть досягати кінцевого результату ваги зображення що може бути в відношенні мінімум 2:1 для TIFF, 16:1 для JPEG 2000 та 2.7 до 1 у PNG

що на жаль значно більше ніж у існуючих методах із використанням нейронних мереж. Найкращі результати відображено на Табл. 4.1 та Табл. 4.2.

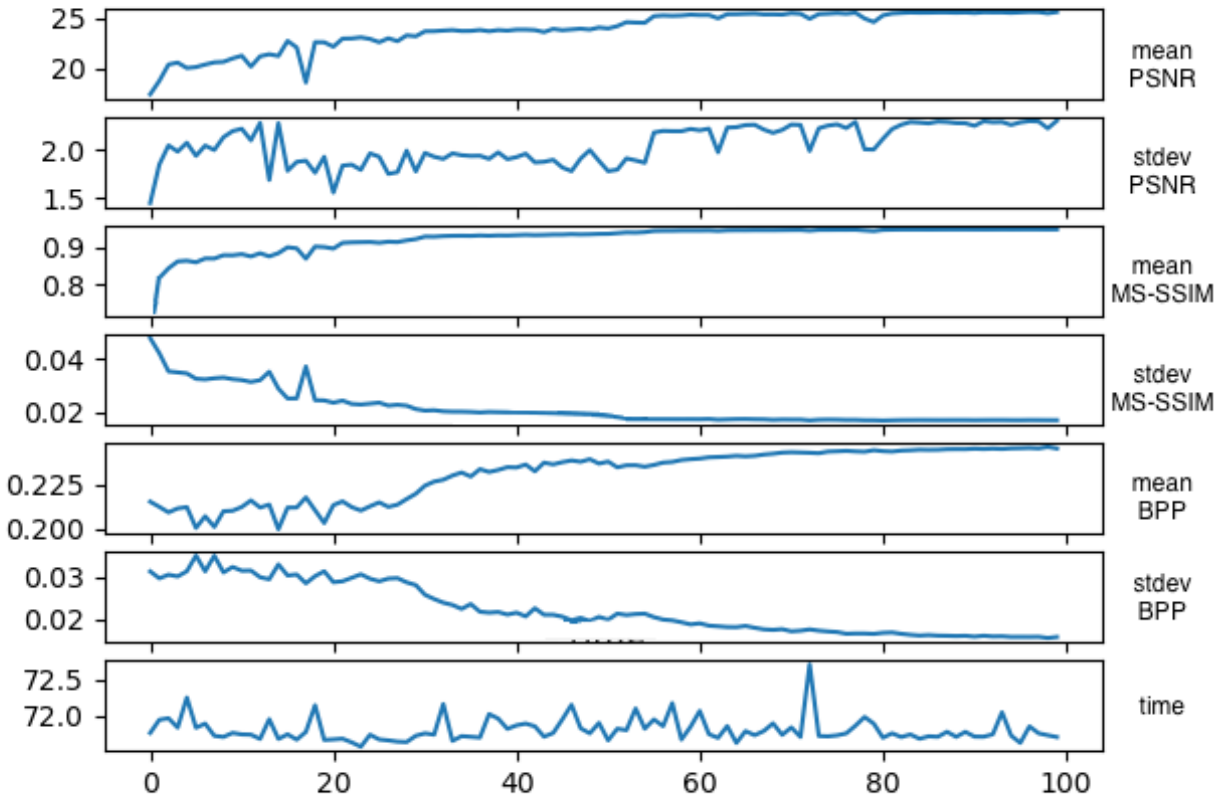

Рис. 4.9 - Predefine Результати тренування

		Mean	
q	CI Type	MS-SSIM/PSNR/BPP (Оригінальна конфігурація)	MS-SSIM/PSNR/BPP (Наша конфігурація)
[3,5,7]	Predefine	0.947 / 24.801 / 0.233	**0.959 / 25.641 / 0.229**

Табл. 4.1 - Результат метрик тренування

STDEV		
MS-SSIM	PSNR	BPP
0.0169	2.299	0.0158

Табл. 4.2 - Похибка метрик тренування

Рис. 4.10 - Оригінальне зображення # 20 Kodak датасету

Рис. 4.11 - Реконструйоване зображення # 20 Kodak датасету

Окрім отриманих результатів із фінальної реконструкції зображенням також хочеться зазначити отримані проміжні результати тренування отримані під час проведення експериментів пов’язаних із зміною параметрів кодувальника та

декодувальник у конфігурації, та виділити їх вплив на саме зображення, в випадку із Рис. 4.10 та Рис. 4.11 ми зменшили значення на En/Decoder Feature параметрі до 0.707 що призвело до появи сітчасто подібних артефактів та значній втраті кольору на перших 20 ітераціях тренування. Даний феномен відродився і на Рис. 4.13 проте крім зміни зазначених гіпер параметрів нейронної мережі ми також підвищили значення на квантувальнику. Рис. 4.13 був отриманий на перших 10 ітераціях процесу тренування.

Рис. 4.12 - Оригінальне зображення # 24 Kodak датасету

Рис. 4.13 - Реконструйоване зображення # 24 Kodak датасету

В той же час коли Рис. 4.14 є результатом тренування на 30 та вище ітераціях. Як ми можемо спостерігати сітчасто підібні артефакти зберігаються проте також наявна дуже сильна різниця у домінуючому кольорі.

Рис. 4.14 - Реконструйоване зображення # 24 Kodak датасету

Загалом різниця у зображеннях що є результатами процесу тренування дозволяє розбити їх на наступні групи у процесі зростання епохи тренування:

- Реконструйовані зображення на перших 30 епохах - мають значну втрату кольору або взагалі чорно-білі.
- Зображення реконструйовані на 30 - 50 епохах - яскраво відрізняються появою кольору проте, кольорова різноманітність дуже низька загалом домінують більше коричневі жовті та червоні тони як це видно на Рис. 4.15

Рис. 4.15 - Реконструйоване зображення на 30 епосі тренування

- Зображення реконструйовані на > 50 епохах - зображення отримує значно вищу деталізацію та починає ставати значно більш схожим на оригінальне

ВИСНОВКИ ДО РОЗДІЛУ 4

У даному розділі було розглянуто отримані після проведення експериментів над досліджуваною технологією результати, описано вплив гіпер параметрів мережі на її фінальний результат виводу у вигляді реконструйованого зображення.

Результати експерименту призвели до отримання наступного росту показників метрик MS-SSIM та PNSR - на 0.012 і 0.84 відповідно. Що дає нам зробити висновок про кращу якість результуючого зображення. Процес тренування проведено більш ефективно зважаючи на те що використаний набір даних на 45% менший та кількістю тренувальних епох менша в 4 рази.

Проте якщо взяти до уваги результати фактичного розміру отриманого зображення, відносно його структурної якості, можна впевнено констатувати, що класичні методи стиснення зображення на даний момент надають нам значно вище значення компресії у порівнянні з існуючими методами що базуються на нейронних мережах. Отже, якщо розглянути практичну сторону розглянутого питання то вкінці кінців треба обирати що потрібно більше - висока якість зображення та гірша компресія чи навпаки трохи гірше зображення при значно більшому коефіцієнті стискання.

ВИСНОВКИ

Дана магістерська робота була присвячена створенню нового методу компресії зображення що базується на основі нейронної мережі задля дослідження можливості використання машинного навчання у сфері де домінують класичні імперативні алгоритми.

У процесі дослідження та роботи над магістерською дисертацією нами була визначено відповідно: мета роботи, предмет та об’єкт проведеного дослідження.

Було проведено аналіз існуючої проблеми компресії зображення та виконано теоретичний огляд теми. Був проведений широкий огляд вже існуючих імперативних алгоритмів компресії зображення зокрема:

- JPEG
- Метод Хаффмана
- DCT

Проведений аналіз існуючих аналогів нейронних мереж для глибокої компресії зображення. Створено оновлену архітектуру методу компресії що спирається на глибокі нейронні мережі. Створений програмний код що написаний на мові програмування Python із використанням відкритої бібліотеки PyTorch та Keras що спеціалізовані на машинному навчанні. Продемонстровано та проаналізовано отримані результати відносно оригінальних даних з подібної архітектури методу компресії.

На основі метрик MS-SSIM, PSNR, що вище на на 0.012 і 0.84 відповідно, відображено високу якість реконструйованого зображення. Процес тренування проведено більш ефективно адже використаний набір даних на 45% менший, а кількістю тренувальних епох менша в 4 рази. Із коефіцієнту компресії, меншого за 40% та значно меншого відносно класичних методів. Зроблено висновок про неефективність даного типу нейронних мереж у поставленій задачі, як інструменту для стиснення, що не було зроблено у оригінальному дослідженні.

Потенційно можливе використання даної моделі мережі для стиснення відео проте не існує гарантії даної гіпотези, що може бути підтверджена у наступних роботах

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Vector vs Raster image [Електронний ресурс] - Режим доступу: https://justcreative.com/wp-content/uploads/2020/05/vector-raster.gif
2. Raster image [Електронний ресурс] - Режим доступу: https://www.printcnx.com/wp-content/uploads/raster.jpg
3. Vector image [Електронний ресурс] - Режим доступу: https://www.printcnx.com/wp-content/uploads/vector.jpg
4. Chakraverty S., Sahoo D.M., Mahato N.R. (2019) McCulloch–Pitts Neural Network Model. In: Concepts of Soft Computing. Springer, Singapore. https://doi.org/10.1007/978-981-13-7430-2_11
5. Shaw G.L. (1986) Donald Hebb: The Organization of Behavior. In: Palm G., Aertsen A. (eds) Brain Theory. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-70911-1_15
6. Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519
7. Marvin Minsky and Seymour A. Papert. 2017. Perceptrons: An Introduction to Computational Geometry. The MIT Press.
8. Aggarwal, Charu C. (2018). Neural Networks and Deep Learning. Springer International Publishing
9. Deep voice architecture [Електронний ресурс] - Режим доступу: https://3.bp.blogspot.com/-bjFYjr2Po2U/WjlNgrInWZI/AAAAAAAACSQ/tfdMAidI8O8EULlJgYoqRWWE9UGIENAkgCLcBGAs/s1600/image1.png
10. Arık, S. Ö., Chrzanowski, M., Coates, A., Diamos, G., Gibiansky, A., Kang, Y., Shoeybi, M. (2017, July). Deep voice: Real-time neural text-to-speech. In International Conference on Machine Learning (pp. 195-204). PMLR.
11. CNN [Електронний ресурс] - Режим доступу: https://scx2.b-cdn.net/gfx/news/hires/2019/threecnnmode.jpg

12. CNN Echocardiogram [Електронний ресурс] - Режим доступу: https://www.rsipvision.com/wp-content/uploads/2018/08/Five-stream-CNN-LSTM.png
13. DCT [Електронний ресурс] - Режим доступу: https://www.researchgate.net/profile/Ivana-Jovanovic-10/publication/253906898/figure/fig1/AS:298198519304194@1448107475630/Example-of-an-overcomplete-set-of-81-2D-DCT-basis-functions-for-patches-of-size-8-8.png
14. Huffman Coding 1[Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-square-01.png
15. Huffman Coding 2[Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-line-01.png
16. Huffman Coding 3[Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-sceem-01.png
17. Huffman Coding 4 [Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-column-01.png
18. Huffman Coding 5 [Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-sceem-02.png
19. Huffman Coding 6 [Електронний ресурс] - Режим доступу: https://www.print-driver.com/wp-content/uploads/2013/02/color-column-02.png
20. Welch T. A. A technique for high-performance data compression // Computer. — 1984. — Т. 6, № 17. — С. 8–19. — doi:10.1109/MC.1984.1659158.
21. Lempel A., Ziv J. Compression of individual sequences via variable-rate coding // IEEE Transactions on Information Theory[en]. — 1978. — Т. 24, № 5. — С. 530–536. — doi:10.1109/TIT.1978.1055934.
22. Johnston, N., Vincent, D., Minnen, D., Covell, M., Singh, S., Chinen, T., ... & Toderici, G. (2018). Improved lossy image compression with priming and spatially adaptive bit rates for recurrent networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4385-4393).

23. Van Oord, A., Kalchbrenner, N., Kavukcuoglu, K. (2016, June). Pixel recurrent neural networks. In International Conference on Machine Learning (pp. 1747-1756). PMLR.
24. Zhou, S., Wu, Y., Ni, Z., Zhou, X., Wen, H., Zou, Y. (2016). Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. arXiv preprint arXiv:1606.06160.
25. Mentzer, F., Agustsson, E., Tschannen, M., Timofte, R., & Van Gool, L. (2018). Conditional probability models for deep image compression. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4394-4402).
26. Li, M., Zuo, W., Gu, S., Zhao, D., Zhang, D. (2018). Learning convolutional networks for content-weighted image compression. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 3214-3223).
27. Liu, H., Chen, T., Guo, P., Shen, Q., Cao, X., Wang, Y., & Ma, Z. (2019). Non-local attention optimized deep image compression. arXiv preprint arXiv:1904.09757.
28. Choi, Y., El-Khamy, M., & Lee, J. (2019). Variable rate deep image compression with a conditional autoencoder. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 3146-3154).
29. Cai, J., & Zhang, L. (2018, October). Deep image compression with iterative non-uniform quantization. In 2018 25th IEEE International Conference on Image Processing *(ICIP)* (pp. 451-455). IEEE.
30. Keras [Електронний ресурс] - Режим доступу: https://keras.io/api/
31. PyTorch [Електронний ресурс] - Режим доступу: https://pytorch.org/docs/stable/index.html
32. Agustsson, E., Timofte, R. (2017). Ntire 2017 challenge on single image super-resolution: Dataset and study. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (c. 126-135).
33. Timofte, R., Agustsson, E., Van Gool, L., Yang, M. H., Zhang, L. (2017). Ntire 2017 challenge on single image super-resolution: Methods and results. In

Proceedings of the IEEE conference on computer vision and pattern recognition workshops (pp. 114-125).

34. Timofte, R., Gu, S., Wu, J., Van Gool, L. (2018). Ntire 2018 challenge on single image super-resolution: Methods and results. In Proceedings of the IEEE conference on computer vision and pattern recognition workshops (C. 852-863).
35. Ignatov, A., Timofte, R., Van Vu, T., Minh Luu, T., X Pham, T., Van Nguyen, C., Jung, C. (2018). Pirm challenge on perceptual image enhancement on smartphones: Report. In Proceedings of the European Conference on Computer Vision (ECCV) Workshops (pp. 0-0).
36. Div2k [Електронний ресурс] - Режим доступу: https://data.vision.ee.ethz.ch/cvl/Div2k/
37. Zhong, Z., Akutsu, H., Aizawa, K. (2020). Channel-Level Variable Quantization Network for Deep Image Compression. arXiv:2007.12619.
38. Ballé, J., Laparra, V., Simoncelli, E. P. (2016). End-to-end optimized image compression. arXiv preprint arXiv:1611.01704.
39. Chen, X., Kingma, D. P., Salimans, T., Duan, Y., Dhariwal, P., Schulman, J., Abbeel, P. (2016). Variational lossy autoencoder. arXiv preprint arXiv:1611.02731.
40. Sun, Q., Ren, Y., Jiao, L., Li, X., Shang, F., Liu, F. (2021). MWQ: Multiscale Wavelet Quantized Neural Networks. arXiv preprint arXiv:2103.05363.
41. Kodak Dataset [Електронний ресурс] - Режим доступу: http://www.cs.albany.edu/~xypan/research/snr/Kodak.html
42. Schiopu, I., & Munteanu, A. (2018). Residual-error prediction based on deep learning for lossless image compression. Electronics Letters, 54(17), 1032-1034.

ДОДАТОК А

Лістинг програми

Київ – 2021

 # component.py

```
import torch
import torch.nn as nn
import torch._utils
import torch.nn.functional as F
from torch.autograd import Variable, Function
import numpy as np
import math
import pdb


def pixel_unshuffle(input, downscale_factor):
        '''
        input: batchSize * c * k*w * k*h
        kdownscale_factor: k
        batchSize * c * k*w * k*h -> batchSize * k*k*c * w * h
        '''
        c = input.shape[1]

        kernel = torch.zeros(size=[downscale_factor * downscale_factor * c,
                                   1, downscale_factor, downscale_factor],
                             device=input.device)
        for y in range(downscale_factor):
        for x in range(downscale_factor):
        kernel[x + y * downscale_factor::downscale_factor * downscale_factor, 0, y, x] = 1
        return F.conv2d(input, kernel, stride=downscale_factor, groups=c)


## Inverse Pixel Shuffle (IPS, part of encoder/decoder)
class InversePixelShuffle(nn.Module):
        def __init__(self, downscale_factor):
        super(InversePixelShuffle, self).__init__()
        self.downscale_factor = downscale_factor
        def forward(self, input):
        '''
        input: batchSize * c * k*w * k*h
        kdownscale_factor: k
        batchSize * c * k*w * k*h -> batchSize * k*k*c * w * h
        '''
        return pixel_unshuffle(input, self.downscale_factor)


class LowerBound(Function):

        @staticmethod
        def forward(ctx, inputs, bound):
        b = torch.ones(inputs.size()) * bound
        b = b.to(inputs.device)
        ctx.save_for_backward(inputs, b)
```

```
        return torch.max(inputs, b)


        @staticmethod
        def backward(ctx, grad_output):
        inputs, b = ctx.saved_tensors

        pass_through_1 = inputs >= b
        pass_through_2 = grad_output < 0

        pass_through = pass_through_1 | pass_through_2
        return pass_through.type(grad_output.dtype) * grad_output, None


## Generalized Divisive Normalization (GDN, part of encoder/decoder)
class GDN(nn.Module):
        """
        Generalized divisive normalization layer.
        y[i] = x[i] / sqrt(beta[i] + sum_j(gamma[j, i] * x[j]))
        """

        def __init__(self,
                ch,
                inverse=False,
                beta_min=1e-6,
                gamma_init=.1,
                reparam_offset=2 ** -18):
        super(GDN, self).__init__()
        device = torch.cuda.current_device()
        self.inverse = inverse
        self.beta_min = beta_min
        self.gamma_init = gamma_init
        self.reparam_offset = torch.FloatTensor([reparam_offset])

        self.build(ch, device)

        def build(self, ch, device):
        self.pedestal = self.reparam_offset ** 2
        self.beta_bound = (self.beta_min + self.reparam_offset ** 2) ** .5
        self.gamma_bound = self.reparam_offset

        # Create beta param
        beta = torch.sqrt(torch.ones(ch) + self.pedestal)
        self.beta = nn.Parameter(beta.to(device))

        # Create gamma param
        eye = torch.eye(ch)
        g = self.gamma_init * eye
        g = g + self.pedestal
        gamma = torch.sqrt(g)
```

```
        self.gamma = nn.Parameter(gamma.to(device))
        self.pedestal = self.pedestal.to(device)

        def forward(self, inputs):
        unfold = False
        if inputs.dim() == 5:
        unfold = True
        bs, ch, d, w, h = inputs.size()
        inputs = inputs.view(bs, ch, d * w, h)

        _, ch, _, _ = inputs.size()

        # Beta bound and reparam
        beta = LowerBound.apply(self.beta, self.beta_bound)
        beta = beta ** 2 - self.pedestal

        # Gamma bound and reparam
        gamma = LowerBound.apply(self.gamma, self.gamma_bound)
        gamma = gamma ** 2 - self.pedestal
        gamma  = gamma.view(ch, ch, 1, 1)

        # Norm pool calc
        norm_ = nn.functional.conv2d(inputs ** 2, gamma, beta)
        norm_ = torch.sqrt(norm_)

        # Apply norm
        if self.inverse:
        outputs = inputs * norm_
        else:
        outputs = inputs / norm_

        if unfold:
        outputs = outputs.view(bs, ch, d, w, h)
        return outputs

## Channel Attention (CA, part of encoder/decoder)
class CALayer(nn.Module):
        def __init__(self, channel, reduction=1):
        super(CALayer, self).__init__()
        # global average pooling: feature --> point
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # feature channel downscale and upscale --> channel weight
        self.conv_du = nn.Sequential(
                nn.Conv2d(channel, channel // reduction, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel, channel // reduction, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // reduction, channel, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // reduction, channel, 1, padding=0, bias=True),
                nn.Softmax()
```

```
        )

        def forward(self, x):
        y = self.avg_pool(x)
        y = self.conv_du(y)
        return x * y

## Residual Channel Attention Block (RCAB, part of encoder/decoder)
class RCAB(nn.Module):
        def __init__(
        self, conv, n_feat, kernel_size, reduction=1,
        bias=True, bn=False, act=nn.ReLU(True), res_scale=1):

        super(RCAB, self).__init__()
        modules_body = []
        for i in range(2):
        modules_body.append(conv(n_feat, n_feat, kernel_size, bias=bias))
        if bn: modules_body.append(nn.BatchNorm2d(n_feat))
        if i == 0: modules_body.append(act)
        modules_body.append(CALayer(n_feat, reduction))
        self.body = nn.Sequential(*modules_body)
        self.res_scale = res_scale

        def forward(self, x):
        res = self.body(x)
        res = self.body(x).mul(self.res_scale)
        res += x
        return res

## Residual Group (RG, part of encoder/decoder)
class ResidualGroup(nn.Module):
        def __init__(self, conv, n_feat, n_blocks, kernel_size, reduction=1, act=nn.ReLU(True),
res_scale=1):
        super(ResidualGroup, self).__init__()
        modules_body = []
        modules_body = [
        RCAB(
                conv, n_feat, kernel_size, reduction, bias=True, bn=False, act=act,
res_scale=res_scale) \
        for _ in range(n_blocks)]
        modules_body.append(conv(n_feat, n_feat, kernel_size))
        self.body = nn.Sequential(*modules_body)

        def forward(self, x):
        res = self.body(x)
        res += x
        return res


## default convolution (part of encoder/decoder)
def default_conv(in_channels, out_channels, kernel_size, bias=True):
```

```
        return nn.Conv2d(
        in_channels, out_channels, kernel_size,
        padding=(kernel_size//2), bias=bias)


## Mean Shift (MS, part of encoder/decoder)
class MeanShift(nn.Conv2d):
        def __init__(self, rgb_range, rgb_mean, rgb_std, sign=-1):
        super(MeanShift, self).__init__(3, 3, kernel_size=1)
        std = torch.Tensor(rgb_std)
        self.weight.data = torch.eye(3).view(3, 3, 1, 1)
        self.weight.data.div_(std.view(3, 1, 1, 1))
        # self.bias.data = sign * rgb_range * torch.Tensor(rgb_mean)
        self.bias.data = sign * torch.Tensor(rgb_mean)
        self.bias.data.div_(std)
        self.requires_grad = False


## Encoder
class Encoder(nn.Module):
        def __init__(self, cfg, **kwargs):
        super(Encoder, self).__init__()

        if cfg['ENC']['ACT'] == 'relu':
        act = nn.ReLU(True)
        elif cfg['ENC']['ACT'] == 'lrelu':
        act = nn.LeakyReLU(0.2, True)
        elif cfg['ENC']['ACT'] == 'identity':
        act = nn.Identity()

        conv = default_conv

        feat_num = cfg['ENC']['FEAT_NUMS']

        # RGB mean for DIV2K
        # rgb_mean = (0.4488, 0.4371, 0.4040)
        # rgb_mean = (0.45659249, 0.43772669, 0.41186953)
        rgb_mean = cfg['DATASET']['MEAN']
        rgb_std = cfg['DATASET']['STD']

        self.sub_mean = MeanShift(255, rgb_mean, rgb_std)

        # define head module
        modules_head = [conv(cfg['IN_CHNS'], feat_num[0], 3)]

        # define body module, four stages
        modules_body = []
        for i in range(4):
        modules_body.append(InversePixelShuffle(2))
        if cfg['ENC']['GDN_FLAG'] == True:
                modules_body.append(GDN(feat_num[i] * 4))
```

```
        modules_body.append(ResidualGroup(conv, feat_num[i] * 4, cfg['ENC']['BLOCK_NUMS'][i],
3, act=act))
        if i < 3:
                modules_body.append(conv(feat_num[i] * 4, feat_num[i + 1], 3))
        else :
                modules_body.append(conv(feat_num[i] * 4, feat_num[i], 3))


        # define tail module
        modules_tail = [conv(feat_num[-1], cfg['CODE_CHNS'], 3)]

        self.head = nn.Sequential(*modules_head)
        self.body = nn.Sequential(*modules_body)
        self.tail = nn.Sequential(*modules_tail)

        def forward(self, x):
        x = self.sub_mean(x)

        x = self.head(x)

        res = self.body(x)

        x = self.tail(res)


        return x




## Decoder
class Decoder(nn.Module):
        def __init__(self, cfg, **kwargs):
        super(Decoder, self).__init__()
        conv = default_conv

        if cfg['DEC']['ACT'] == 'relu':
        act = nn.ReLU(True)
        elif cfg['DEC']['ACT'] == 'lrelu':
        act = nn.LeakyReLU(0.05, True)
        elif cfg['DEC']['ACT'] == 'identity':
        act = nn.Identity()

        feat_num = cfg['DEC']['FEAT_NUMS']

        # RGB mean for DIV2K
        # rgb_mean = (0.4488, 0.4371, 0.4040)

        rgb_mean = cfg['DATASET']['MEAN']
        rgb_std = cfg['DATASET']['STD']
        self.add_mean = MeanShift(255, rgb_mean, rgb_std, 1)
```

```
        # define head module
        modules_head = [conv(cfg['CODE_CHNS'], feat_num[0], 3)]


        # define body module
        modules_body = []
        for i in range(4):
        if i < 3:
                modules_body.append(conv(feat_num[i], feat_num[i + 1] * 4, 3))
                modules_body.append(ResidualGroup(conv, feat_num[i + 1] * 4,
cfg['DEC']['BLOCK_NUMS'][i], 3, act=act))
                if cfg['DEC']['GDN_FLAG'] == True:
                modules_body.append(GDN(feat_num[i + 1] * 4))
        else :
                modules_body.append(conv(feat_num[i], feat_num[i] * 4, 3))
                modules_body.append(ResidualGroup(conv, feat_num[i] * 4,
cfg['DEC']['BLOCK_NUMS'][i], 3, act=act))
                if cfg['DEC']['GDN_FLAG'] == True:
                modules_body.append(GDN(feat_num[i] * 4))

        modules_body.append(nn.PixelShuffle(2))

        # define tail module
        modules_tail = [conv(feat_num[-1], cfg['IN_CHNS'], 3)]

        self.head = nn.Sequential(*modules_head)
        self.body = nn.Sequential(*modules_body)
        self.tail = nn.Sequential(*modules_tail)

        def forward(self, x):

        x = self.head(x)

        res = self.body(x)

        x = self.tail(res)

        x = self.add_mean(x)

        return x


## 3D Masked Convolution (part of entropy model)
class MaskedConv3d(torch.nn.Conv3d):
        def __init__(self, mask_type, *args, **kwargs):
        super(MaskedConv3d, self).__init__(*args, **kwargs)
        assert mask_type in {'hollow', 'solid'}
        self.register_buffer('mask', self.weight.data.clone())
        _, _, kZ, kH, kW = self.weight.size()
        self.mask.fill_(1)
```

```
        center_idx = kZ * kH * kW // 2
        cur_idx = 0
        for i in range(kZ):
        for j in range(kH):
                for k in range(kW):
                if cur_idx < center_idx + (mask_type == 'solid'):
                self.mask[:, :, i, j, k] = 1
                else:
                self.mask[:, :, i, j, k] = 0
                cur_idx += 1

        def forward(self, x):
        self.weight.data *= self.mask
        return super(MaskedConv3d, self).forward(x)[:, :, :, :, :]


## 3D Entropy Context Estimate Model
class ContextEstimater3d(torch.nn.Module):
        def __init__(self, fm_centers_n, internal_dps):
        super(ContextEstimater3d, self).__init__()
        self.fm_centers_n = fm_centers_n
        self.kernel_n = (3, 3, 3)
        # self.pad = torch.nn.ConstantPad3d((1, 1, 1, 1, 1, 1), 1)
        self.pad_n = (1, 1, 1)
        self.stride_n = (1, 1, 1)
        self.internal_dps = internal_dps
        self.conv_bias = True
        self.active_func = torch.nn.functional.relu

        self.first_layer = MaskedConv3d('hollow', 1, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)

        self.middle_layer1 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)
        self.middle_layer2 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)

        self.middle_layer3 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)
        self.middle_layer4 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)

        self.middle_layer5 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
                                        self.kernel_n, self.stride_n, self.pad_n,
                                        bias=self.conv_bias)
        self.middle_layer6 = MaskedConv3d('solid', self.internal_dps, self.internal_dps,
```

```
                                self.kernel_n, self.stride_n, self.pad_n,
                                bias=self.conv_bias)

        self.final_layer = MaskedConv3d('solid', self.internal_dps, self.fm_centers_n,
                                self.kernel_n, self.stride_n, self.pad_n,
                                bias=self.conv_bias)

        def forward(self, input_tensor):
        out1 = self.first_layer(input_tensor)
        out1 = self.active_func(out1)

        out2 = self.middle_layer1(out1)
        out2 = self.active_func(out2)
        out2 = self.middle_layer2(out2)
        out2 = out2+out1

        out3 = self.final_layer(out2)
        out3 = self.active_func(out3)

        return out3


## GMM Quantizer
class GMMQuantizer(nn.Module):
        def __init__(self, num_of_mean, std_list, pi_list, mean_list=None):
        super(GMMQuantizer, self).__init__()
        self.num_of_mean = num_of_mean
        self.sigma = 1.0
        self.hard_sigma = 1e4
        self.div = 1e-3

        if mean_list == None:
        mean_list = np.linspace(-(num_of_mean//2), num_of_mean//2, num_of_mean)
        mean_list = mean_list.astype(np.float32)
        mean = torch.from_numpy(mean_list)
        self.mean = torch.nn.Parameter(mean)
        else:
        mean = torch.from_numpy(np.array(mean_list)).float()
        self.mean = torch.nn.Parameter(mean)

        std = torch.from_numpy(np.array(std_list)).float()
        log_std = torch.log(std)
        self.log_std = torch.nn.Parameter(log_std)
        self.std = None

        pi = torch.from_numpy(np.array(pi_list)).float()
        log_pi = torch.log(pi)
        self.log_pi = torch.nn.Parameter(log_pi)
        self.norm_pi = None
```

```
        def forward(self, input_tensor):

        self._get_norm_pi()
        self._get_std()

        dist = self._get_dist_between_mean(input_tensor)
        dist = dist / 2 / (self.std * self.std + self.div)
        phi_soft = self._weighted_softmax(-self.sigma * dist)
        phi_hard = self._weighted_softmax(-self.hard_sigma * dist)
        symbols_hard = torch.argmax(phi_hard, dim=4)
        symbols_hard = symbols_hard.unsqueeze(4)
        shape = symbols_hard.shape
        one_hot = torch.cuda.FloatTensor(shape[0], shape[1], shape[2], shape[3],
self.num_of_mean)
        one_hot.zero_()
        one_hot.scatter_(4, symbols_hard, 1)

        softout = torch.sum(phi_soft * self.mean, dim=4)
        hardout = torch.sum(one_hot * self.mean, dim=4)


        mid_tensor_q = softout + (hardout - softout).detach()
        return mid_tensor_q, symbols_hard

        def _get_dist_between_mean(self, input_tensor):
        dist = torch.unsqueeze(input_tensor, 4) - self.mean
        dist = torch.abs(dist)
        dist = torch.mul(dist, dist)
        return dist

        def _weighted_softmax(self, x):
        # x is b x c x h x w x L
        maxes, _ = torch.max(x, dim=4)
        x_exp = torch.exp(x - maxes.unsqueeze(-1)) / torch.sqrt(math.pi * 2 * (self.std * self.std +
self.div)) * self.norm_pi
        # x_exp_sum is b x c x h x w
        x_exp_sum = torch.sum(x_exp, 4)
        return x_exp / x_exp_sum.unsqueeze(4)

        def _get_norm_pi(self):
        min_log_pi = torch.min(self.log_pi)
        mixing_proportions = torch.exp(self.log_pi - min_log_pi)
        self.norm_pi = mixing_proportions / torch.sum(mixing_proportions)
        return

        def _get_std(self):
        self.std = torch.exp(self.log_std)
        return

        def get_prob(self, x):
        """
```

```
        x: b x c x h x w
        mean: L
        std: L
        pi: L
        prob: b x c x h x w x L
        out: b x c x h x w x 1
        """
        self._get_norm_pi()
        self._get_std()
        prob = torch.exp(-(x.unsqueeze(4) - self.mean) * (x.unsqueeze(4) - self.mean)/ 2 / (self.std *
self.std + self.div)) * self.norm_pi / torch.sqrt(math.pi * 2 * (self.std * self.std + self.div))
        out = torch.sum(prob, 4)
        return out

# Channel attention based learning for channel importance vector
class ChannelImportance(nn.Module):
        def __init__(self, channel, reduction=1):
        super(ChannelImportance, self).__init__()
        # global average pooling: feature --> point
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        # feature channel downscale and upscale --> channel weight
        self.conv_du = nn.Sequential(
                nn.Conv2d(channel, channel // reduction, 1, padding=0, bias=True),
                nn.ReLU(inplace=True),
                nn.Conv2d(channel // reduction, channel, 1, padding=0, bias=True),
                nn.Sigmoid()
        )

        def forward(self, x):
        y = self.avg_pool(x)
        y = self.conv_du(y)
        return y
```

 # trainer.py

```
import torch
import torch.utils.data as data
import torch.optim.lr_scheduler as LS
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from skimage.metrics import structural_similarity as ssim
from torchvision import transforms
from torch.utils.tensorboard import SummaryWriter
import matplotlib.pylab as plt
import numpy as np
import os.path as osp
import os as os
import time
import pdb
import calendar;
```

```
from operator import itemgetter as i
from functools import cmp_to_key

import tensorflow as tf

from pytorch_msssim import msssim
from config import config
from config import update_config, create_logger
import dataset
from component import *
from torchvision.utils import save_image

class Trainer():
        def __init__(self, cfg, logger, model_dir, tensorboard_log_dir, imp, **kwargs):
        torch.cuda.set_device(cfg['GPU_DEVICE'])
        self.imp = imp
        self.metrics_train_disstortion = list()
        self.metrics_train_entropy = list()
        self.metrics_train_batch_time = list()

        self.metrics_eval_psnr = list()
        self.metrics_eval_msssim = list()
        self.metrics_eval_bpp = list()
        self.metrics_eval_test_time = list()

        self.nums_epoch = cfg['TRAIN']['NUM_EPOCH']
        self.epoch = 0
        self.wr_train_idx = 1
        self.wr_eval_idx = 1
        self.scale = 16
        self.max_psnr = 0.
        self.max_ssim = 0.
        self.its_bpp = 24.

        self.log = logger
        self.alpha = cfg['TRAIN']['ALPHA']
        self.beta = cfg['TRAIN']['BETA']
        self.print_freq = cfg['PRINT_FREQ']
        self.quant_level = cfg['QUA_LEVELS']

        self.model_dir = model_dir
        self.writer = SummaryWriter(tensorboard_log_dir)

        self.encoder = Encoder(cfg).cuda()
        self.decoder = Decoder(cfg).cuda()

        self.quantizer = nn.ModuleList()
        self.entropy = nn.ModuleList()

        self.clip_value = cfg['TRAIN']['CLIP_VALUE']
```

```
        self.part = [0]
        [self.part.append(int(x*cfg['CODE_CHNS']) + self.part[i]) for i, x in
enumerate(cfg['ENP']['PART'])]

        for i in range(len(cfg['ENP']['PART'])):
        self.quantizer.append(GMMQuantizer(cfg['QUA_LEVELS'][i], cfg['QUA']['STD'][i],
cfg['QUA']['PI'][i]).cuda())
        self.entropy.append(ContextEstimater3d(cfg['QUA_LEVELS'][i],
cfg['ENP']['FEAT_NUMS'][i]).cuda())


        self.code_nums = cfg['CODE_CHNS']
        self.solver_enc = optim.Adam(self.encoder.parameters(), lr=cfg['TRAIN']['LR_ENC'])

        Q_params = list()
        for i in range(len(cfg['QUA_LEVELS'])):
        Q_params.append({'params': self.quantizer[i].mean,          'lr': cfg['TRAIN']['LR_QUA']    })
        Q_params.append({'params': self.quantizer[i].log_pi,  'lr': cfg['TRAIN']['LR_QUA'] * 0.2})
        Q_params.append({'params': self.quantizer[i].log_std, 'lr': cfg['TRAIN']['LR_QUA'] * 0.2})
        self.solver_qua = optim.Adam(Q_params)


        self.solver_etp = optim.Adam(self.entropy.parameters(), lr=cfg['TRAIN']['LR_ENP'])
        self.solver_dec = optim.Adam(self.decoder.parameters(), lr=cfg['TRAIN']['LR_DEC'])


        self.scheduler_enc = LS.MultiStepLR(self.solver_enc,
milestones=cfg['TRAIN']['MILESTONES'], gamma=cfg['TRAIN']['GAMMA'])
        self.scheduler_qua = LS.MultiStepLR(self.solver_qua,
milestones=cfg['TRAIN']['MILESTONES'], gamma=cfg['TRAIN']['GAMMA'])
        self.scheduler_etp = LS.MultiStepLR(self.solver_etp,
milestones=cfg['TRAIN']['MILESTONES'], gamma=cfg['TRAIN']['GAMMA'])
        self.scheduler_dec = LS.MultiStepLR(self.solver_dec,
milestones=cfg['TRAIN']['MILESTONES'], gamma=cfg['TRAIN']['GAMMA'])


        train_transform = transforms.Compose([
        transforms.RandomCrop((cfg['DATASET']['PATCH_SIZE'], cfg['DATASET']['PATCH_SIZE'])),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ToTensor(),])
        train_set = dataset.ImageFolder(root=cfg['DATASET']['TRAIN_SET'],
transform=train_transform)
        self.train_loader = data.DataLoader(dataset=train_set,
        batch_size=cfg['TRAIN']['BATCH_SIZE'], shuffle=True, num_workers=cfg['WORKERS'])

        test_transform = transforms.Compose([transforms.ToTensor(),])
        test_set = dataset.ImageFolder(root=cfg['DATASET']['TEST_SET'],
transform=test_transform)
        self.test_loader = data.DataLoader(dataset=test_set,
```

```
        batch_size=1, shuffle=False, num_workers=cfg['WORKERS'])
        self.entropy_loss = nn.CrossEntropyLoss()

        def train(self):
        self.log.info('------------------------------Training------------------------------')

        self._train_init()

        loss_distortions_train = list()
        loss_entropies_train = list()
        batch_times_train = list()
        for batch, (img, _) in enumerate(self.train_loader):

        batch_t0 = time.time()
        batch_t1 = time.time()

        self._solver_init()

        # float image 0~1
        img = img.cuda()

        # feat: 32 x 64 x 8 x 8
        feat = self.encoder(img)

        # bc: 32 x 64 x 8 x 8 (int)
        feat_new = self._order_feat(feat)

        bc_list = list()
        idx_list = list()
        prob_list = list()
        # loss_gmm = 0
        for i in range(len(self.part) - 1):

                bc, idx = self.quantizer[i](feat_new[:, self.part[i]:self.part[i + 1], :, :])
                # loss_gmm -=
torch.mean(torch.log(self.quantizer[i].get_prob(feat_new[:,self.part[i]:self.part[i+1],:,:])))
                bc_list.append(bc)
                idx_list.append(idx.squeeze(-1))
                prob_list.append(self.entropy[i](bc_list[i].unsqueeze(1)))

        # loss_gmm = loss_gmm / (len(self.part)-1)

        bc = bc_list[0]
        for i in range(1, len(self.part) - 1):
                bc = torch.cat((bc, bc_list[i]), dim=1)

        bc_reorder = self._reorder_code(bc)

        re = self.decoder(bc_reorder)

        img255 = img.mul(255).add_(0.5).clamp_(0, 255).floor()
```

```
        re_ = re.mul(255).add_(0.5).clamp_(0, 255)
        re255 = (re_.floor() - re_).detach() + re_

        loss_distortion = 1 - msssim(img255, re255)


        loss_entropy = 0

        for i in range(len(idx_list)):
                loss_entropy += self.entropy_loss(prob_list[i], idx_list[i])

        loss_entropy = loss_entropy / len(idx_list)

        # loss = self.alpha * loss_distortion + loss_entropy + self.beta * loss_gmm
        loss = self.alpha * loss_distortion + loss_entropy

        loss.backward()

        self._solver_update()

        if (batch + 1) % (self.print_freq) == 0:

                self.writer.add_scalar('loss_dis', loss_distortion, self.wr_train_idx)
                self.writer.add_scalar('loss_en', loss_entropy, self.wr_train_idx)
                self.wr_train_idx += 1

                batch_t1 = time.time()
                for i in range(len(self.part) - 1):
                self.log.info('Quantizer[{}/{}]: '.format(i + 1, len(self.part) - 1) + ' mean:\t' +
str(np.round(self.quantizer[i].mean.data.cpu().numpy(),4)))
                self.log.info('Quantizer[{}/{}]: '.format(i + 1, len(self.part) - 1) + ' std:\t'  +
str(np.round(self.quantizer[i].std.data.cpu().numpy(),4)))
                self.log.info('Quantizer[{}/{}]: '.format(i + 1, len(self.part) - 1) + ' pi:\t'   +
str(np.round(self.quantizer[i].norm_pi.data.cpu().numpy(),4)))

                self.log.info('Epoch[{}/{}]({}/{}):\t'.format(self.epoch + 1, self.nums_epoch, batch + 1,
len(self.train_loader)))

        loss_diss = loss_distortion.item()
        loss_en = loss_entropy.item()
        batch_time = batch_t1 - batch_t0

        self.log.info('\t Loss_dis: {:.3f};\t Loss_etp: {:.3f};\t Batch time: {:.3f} sec.'.format(loss_diss,
loss_en, batch_time))

        self.metrics_train_disstortion.append(loss_diss)
        self.metrics_train_entropy.append(loss_en)
        self.metrics_train_batch_time.append(batch_time)

        self.epoch += 1
```

```
def eval(self):
self.log.info('-----------------------------Evaluation-----------------------------')
eval_t0 = time.time()

self._eval_init()

ssim_mean = []
psnr_mean = []
bpp_mean = []

en_idx=0

eval_results = []

for _, (img, _) in enumerate(self.test_loader):
en_idx+=1
# float image 0~1
img = img.cuda()

_, _, h, w = img.size()
# feat: 32 x 64 x 8 x 8
feat = self.encoder(img)

# bc: 32 x 64 x 8 x 8 (int)
feat_new = self._order_feat(feat)

bc_list = list()
idx_list = list()
prob_list = list()
for i in range(len(self.part) - 1):
        bc, idx = self.quantizer[i](feat_new[:,self.part[i]:self.part[i + 1],:,:])
        bc_list.append(bc)
        idx_list.append(idx.squeeze(-1))
        prob_list.append(self.entropy[i](bc_list[i].unsqueeze(1)))

bc = bc_list[0]
for i in range(1, len(self.part) - 1):
        bc = torch.cat((bc, bc_list[i]), dim=1)

bc_reorder = self._reorder_code(bc)

re = self.decoder(bc_reorder)

bpp_item = 0
for i in range(len(idx_list)):
        bpp_item += F.cross_entropy(prob_list[i], idx_list[i], reduction='sum')


bpp_item = bpp_item * torch.log2(torch.tensor(2.71828)).cuda() / h / w
bpp_mean.append(bpp_item.item())
```

```
        img255 = img.squeeze().mul(255).add_(0.5).clamp_(0, 255).floor()
        re255 = re.squeeze().mul(255).add_(0.5).clamp_(0, 255).floor()

        psnr_item = self.compute_psnr(img255.unsqueeze(0), re255.unsqueeze(0))
        psnr_mean.append(psnr_item.item())

        ssim_item = msssim(img255.unsqueeze(0), re255.unsqueeze(0))
        ssim_mean.append(ssim_item.item())

        eval_results.append({
                u'img': re.unsqueeze(0),
                u'psnr': psnr_item.item(),
                u'mssim':ssim_item.item(),
                u'bpp':bpp_item.item()
        })

        eval_t1 = time.time()

        self.writer.add_scalar('BPP', np.mean(bpp_mean), self.wr_eval_idx)
        self.writer.add_scalar('MS-SSIM', np.mean(ssim_mean), self.wr_eval_idx)

        self.wr_eval_idx += 1

        self.metrics_eval_psnr.append(np.mean(psnr_mean))
        self.metrics_eval_msssim.append(np.mean(ssim_mean))
        self.metrics_eval_bpp.append(np.mean(bpp_mean))
        self.metrics_eval_test_time.append(eval_t1 - eval_t0)

        self.log.info('Mean PSNR: {:.3f};\t Mean MS-SSIM: {:.5f};\t Mean BPP: {:.5f};\t Test time:
{:.3f} sec.'.
        format(np.mean(psnr_mean), np.mean(ssim_mean), np.mean(bpp_mean), eval_t1 -
eval_t0))

        if  np.mean(ssim_mean) > self.max_ssim:

        self.save_checkpoint('best.pth')

        self.max_psnr = np.mean(psnr_mean)
        self.max_ssim = np.mean(ssim_mean)
        self.its_bpp = np.mean(bpp_mean)

        self.log.info(', '.join(str(x) for x in ssim_mean) )

        self.print_best_result(eval_results)

        self.log.info('Best PSNR: {:.3f};\t Best MS-SSIM: {:.5f};\t Its BPP: {:.5f}.'.
        format(self.max_psnr, self.max_ssim, self.its_bpp))

        def save_checkpoint(self, model_name):
        raise NotImplementedError("Must inherit from Trainer.")
```

```
        def load_checkpoint(self, date, model_name):
        raise NotImplementedError("Must inherit from Trainer.")

        def update_lr(self):
        raise NotImplementedError("Must inherit from Trainer.")

        def _train_init(self):
        raise NotImplementedError("Must inherit from Trainer.")

        def _eval_init(self):
        raise NotImplementedError("Must inherit from Trainer.")

        def _solver_init(self):
        raise NotImplementedError("Must inherit from Trainer.")

        def _solver_update(self):
        raise NotImplementedError("Must inherit from Trainer.")

        def _order_feat(self, feat):
        raise NotImplementedError("Must inherit from Trainer.")

        def _reorder_code(self, bc):
        raise NotImplementedError("Must inherit from Trainer.")

        def compute_psnr(self, img, re):
        img = img.squeeze()
        re = re.squeeze()
        mse = torch.mean((img - re) ** 2)
        psnr = 10 * (2 * torch.log10(torch.tensor(255.).cuda()) - torch.log10(mse))
        return psnr

        def rgb2yCbCr(self, img_rgb):
        im_flat = img_rgb.contiguous().view(-1, 3).float()
        mat = torch.tensor([[65.481, 128.553, 24.966],
        [-37.797, -74.203, 112.0],
        [112.0, -93.786, -18.214]]).cuda()
        bias = torch.tensor([16.0, 128.0, 128.0]).cuda()
        img_yCbCr = torch.round(im_flat.mm(mat.T) * 1.0 / 255  + bias)
        img_yCbCr = img_yCbCr.view(input_im.shape[0], input_im.shape[1], 3)
        return img_yCbCr

        def save(self, timestamp, restructured):
        save_image(restructured.clone().detach(),'result/image_re_{0}.png'.format(timestamp))

        def metrics_print(self, cfg):
        epoches = list(range(0, self.nums_epoch))

        training, (training_disstortion, training_entropy, training_batch_time) = plt.subplots(3,
sharex=True)
        evaluation, (eval_psnr, eval_msssim, eval_bpp, eval_test_time ) = plt.subplots(4,
sharex=True)
```

```
        training.suptitle('{0} Training metrics'.format(self.imp))
        evaluation.suptitle('{0} Evaluation metrics'.format(self.imp))

        training_disstortion.set_title("Loss Disstortion")
        training_entropy.set_title("Loss Entropy")
        training_batch_time.set_title("Batch Time")

        eval_psnr.set_title("PSNR")
        eval_msssim.set_title("MS-SSIM")
        eval_bpp.set_title("BPP")
        eval_test_time.set_title("Time")

        training_disstortion.plot(epoches, self.metrics_train_disstortion)
        training_entropy.plot(epoches, self.metrics_train_entropy)
        training_batch_time.plot(epoches, self.metrics_train_batch_time)

        eval_psnr.plot(epoches, self.metrics_eval_psnr)
        eval_msssim.plot(epoches, self.metrics_eval_msssim)
        eval_bpp.plot(epoches, self.metrics_eval_bpp)
        eval_test_time.plot(epoches, self.metrics_eval_test_time)

        timestamp = calendar.timegm(time.gmtime())
        training.savefig('{1}/train_{0}_.png'.format(timestamp,self.imp))
        evaluation.savefig('{1}/eval_{0}_.png'.format(timestamp,self.imp))

        def print_best_result(self, result_dict):
        def cmp(a, b):
        return (a > b) - (a < b)

        def multikeysort(items, columns):
        comparers = [
                ((i(col[1:].strip()), -1) if col.startswith('-') else (i(col.strip()), 1))
                for col in columns
        ]

        def comparer(left, right):
                comparer_iter = (
                cmp(fn(left), fn(right)) * mult
                for fn, mult in comparers
                )
                return next((result for result in comparer_iter if result), 0)
        return sorted(items, key=cmp_to_key(comparer))

        best = multikeysort(result_dict, ['mssim'])[-1]
        timestamp = calendar.timegm(time.gmtime())

        self.log.info("Best image MSSIM: {:.3f}; PSNR: {:.5f}; BPP: {:.3f}; Timestamp:
{:.10f}".format(best['mssim'], best['psnr'], best['bpp'], timestamp))
        self.save(timestamp, best['img'])
```

```
class PredefineTrainer(Trainer):
        def __init__(self, cfg, logger, model_dir, tensorboard_log_dir,imp, **kwargs):
        super(PredefineTrainer, self).__init__(cfg, logger, model_dir, tensorboard_log_dir, imp,
**kwargs)


        def save_checkpoint(self, model_name):
        self.log.info('------------------------------Save Cpt------------------------------')

        ckp_path = osp.join(self.model_dir, model_name)

        self.log.info('Save checkpoint: %s' % ckp_path)

        obj = {
        'encoder': self.encoder.state_dict(),
        'quantizer': self.quantizer.state_dict(),
        'decoder': self.decoder.state_dict(),
        'entropy': self.entropy.state_dict(),
        'epoch': self.epoch
        }
        torch.save(obj, ckp_path)
        self.log.info('Save the trained model successfully.')

        def load_checkpoint(self, date, model_name):
        self.log.info('------------------------------Load Cpt------------------------------')
        ckp_path = osp.join(self.model_dir[:-12] + date, model_name)
        try:
        obj = torch.load(ckp_path, map_location=lambda storage, loc: storage.cuda())
        self.log.info('Load checkpoint: %s' % ckp_path)
        except IOError:
        self.log.error('No checkpoint: %s!' % ckp_path)
        return
        self.encoder.load_state_dict(obj['encoder'])
        self.quantizer.load_state_dict(obj['quantizer'])
        self.decoder.load_state_dict(obj['decoder'])
        self.entropy.load_state_dict(obj['entropy'])
        self.epoch = obj['epoch']
        self.log.info('The loaded model has been trained for %d epoch(s).' % self.epoch)


        def update_lr(self):
        self.scheduler_enc.step()
        self.scheduler_qua.step()
        self.scheduler_dec.step()
        self.scheduler_etp.step()

        def _train_init(self):
        self.encoder.train()
        for param in self.encoder.parameters():
        param.requires_grad = True
```

```
self.quantizer.train()
for param in self.quantizer.parameters():
param.requires_grad = True

self.entropy.train()
for param in self.entropy.parameters():
param.requires_grad = True

self.decoder.train()
for param in self.decoder.parameters():
param.requires_grad = True


def _eval_init(self):
self.encoder.eval()
for param in self.encoder.parameters():
param.requires_grad = False

self.quantizer.eval()
for param in self.quantizer.parameters():
param.requires_grad = False

self.entropy.eval()
for param in self.entropy.parameters():
param.requires_grad = False

self.decoder.eval()
for param in self.decoder.parameters():
param.requires_grad = False

def _solver_init(self):
self.solver_qua.zero_grad()
self.solver_dec.zero_grad()
self.solver_etp.zero_grad()
self.solver_enc.zero_grad()

def _solver_update(self):
torch.nn.utils.clip_grad_value_(self.encoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.decoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.quantizer.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.entropy.parameters(), self.clip_value)

self.solver_enc.step()
self.solver_qua.step()
self.solver_dec.step()
self.solver_etp.step()

def _order_feat(self, feat):
return feat

def _reorder_code(self, bc):
```

```
        return bc


class RETrainer(Trainer):
        def __init__(self, cfg, logger, model_dir, tensorboard_log_dir, imp, **kwargs):
        super(RETrainer, self).__init__(cfg, logger, model_dir, tensorboard_log_dir,imp , **kwargs)
        self.imp_value = torch.ones(cfg['CODE_CHNS']).cuda()
        self.order = torch.randperm(cfg['CODE_CHNS']).cuda()
        _, self.reorder = torch.sort(self.order)
        self.log.info('initialized imp order: ')
        self.log.info(self.order)
        imp_transform = transforms.Compose([
        transforms.RandomCrop((256, 256)),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.ToTensor(),])
        imp_set = dataset.ImageFolder(root=cfg['DATASET']['TRAIN_SET'],
transform=imp_transform)
        self.imp_loader = data.DataLoader(dataset=imp_set, batch_size=1, shuffle=True,
num_workers=cfg['WORKERS'])
        self.end_batch = cfg['RE_END_BATCH']


        def save_checkpoint(self, model_name):
        self.log.info('------------------------------Save Cpt------------------------------')

        ckp_path = osp.join(self.model_dir, model_name)

        self.log.info('Save checkpoint: %s' % ckp_path)

        obj = {
        'encoder': self.encoder.state_dict(),
        'quantizer': self.quantizer.state_dict(),
        'decoder': self.decoder.state_dict(),
        'entropy': self.entropy.state_dict(),
        'epoch': self.epoch
        }
        torch.save(obj, ckp_path)
        self.log.info('Save the trained model successfully.')

        def load_checkpoint(self, date, model_name):
        self.log.info('------------------------------Load Cpt------------------------------')
        ckp_path = osp.join(self.model_dir[:-12] + date, model_name)
        try:
        obj = torch.load(ckp_path, map_location=lambda storage, loc: storage.cuda())
        self.log.info('Load checkpoint: %s' % ckp_path)
        except IOError:
        self.log.error('No checkpoint: %s!' % ckp_path)
        return
        self.encoder.load_state_dict(obj['encoder'])
        self.quantizer.load_state_dict(obj['quantizer'])
```

```
self.decoder.load_state_dict(obj['decoder'])
self.entropy.load_state_dict(obj['entropy'])
self.epoch = obj['epoch']
self.log.info('The loaded model has been trained for %d epoch(s).' % self.epoch)


def update_lr(self):
self.scheduler_enc.step()
self.scheduler_qua.step()
self.scheduler_dec.step()
self.scheduler_etp.step()

def _train_init(self):
self.encoder.train()
for param in self.encoder.parameters():
param.requires_grad = True

self.quantizer.train()
for param in self.quantizer.parameters():
param.requires_grad = True

self.entropy.train()
for param in self.entropy.parameters():
param.requires_grad = True

self.decoder.train()
for param in self.decoder.parameters():
param.requires_grad = True


def _eval_init(self):
self.encoder.eval()
for param in self.encoder.parameters():
param.requires_grad = False

self.quantizer.eval()
for param in self.quantizer.parameters():
param.requires_grad = False

self.entropy.eval()
for param in self.entropy.parameters():
param.requires_grad = False

self.decoder.eval()
for param in self.decoder.parameters():
param.requires_grad = False

def _solver_init(self):
self.solver_qua.zero_grad()
self.solver_dec.zero_grad()
self.solver_etp.zero_grad()
```

```
self.solver_enc.zero_grad()

def _solver_update(self):
torch.nn.utils.clip_grad_value_(self.encoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.decoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.quantizer.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.entropy.parameters(), self.clip_value)

self.solver_enc.step()
self.solver_qua.step()
self.solver_dec.step()
self.solver_etp.step()

def _order_feat(self, feat):
return feat[:, self.order, :, :]

def _reorder_code(self, bc):
return bc[:,self.reorder,:,:]

def re_based_get_imp(self):
self.log.info('-----------------Get Importance Based on RE------------------')
imp_t0 = time.time()

self._eval_init()

ssim_mean = torch.zeros(self.code_nums, self.end_batch)
l2_err_mean = torch.zeros(self.code_nums, self.end_batch)
bpp_mean = torch.zeros(self.code_nums, self.end_batch)

for batch, (img, _) in enumerate(self.imp_loader):
if batch == self.end_batch:
        break

# float image 0~1
img = img.cuda()

# feat: 32 x 64 x 8 x 8
feat = self.encoder(img)
# bc: 32 x 64 x 8 x 8 (int)
feat_new = self._order_feat(feat)


bc_list = list()
idx_list = list()
prob_list = list()
for i in range(len(self.part)-1):
        bc, idx = self.quantizer[i](feat_new[:,self.part[i]:self.part[i+1],:,:])
        bc_list.append(bc)
        idx_list.append(idx.squeeze(-1))
        prob_list.append(self.entropy[i](bc_list[i].unsqueeze(1)))
```

```
        bc = bc_list[0]
        for i in range(1, len(self.part)-1):
                bc = torch.cat((bc, bc_list[i]), dim=1)


        bc_reorder = self._reorder_code(bc)

        re = self.decoder(bc_reorder)

        _, ch, _, _ = bc_reorder.size()
        _, _, h, w = img.size()

        for i in range(ch):
                bc_reorder_new = bc_reorder.clone()
                bc_reorder_new[:,i,:,:] = 0
                re_ch = self.decoder(bc_reorder_new)
                ssim_mean[i, batch] = msssim(re, re_ch)
                l2_err_mean[i, batch] = torch.mean((re-re_ch)**2)

        ssim_mean = torch.mean(ssim_mean, dim=1)
        l2_err_mean = torch.mean(l2_err_mean, dim=1)
        bpp_mean = torch.mean(bpp_mean, dim=1)
        bpp_mean = bpp_mean[self.reorder]
        self.imp_value = (1 - ssim_mean) * 200
        self.log.info("Current importance value: ")
        self.log.info(self.imp_value)
        _, self.order = torch.sort(self.imp_value)
        self.log.info("Current importance order: ")
        self.log.info(self.order)
        _, self.reorder = torch.sort(self.order)
        imp_t1 = time.time()
        self.log.info('Update channel importance time: {:.3f} sec.'.format(imp_t1 - imp_t0))


class SETrainer(Trainer):
        def __init__(self, cfg, logger, model_dir, tensorboard_log_dir,imp , **kwargs):
        super(SETrainer, self).__init__(cfg, logger, model_dir, tensorboard_log_dir, imp, **kwargs)
        self.se_based = ChannelImportance(cfg['CODE_CHNS']).cuda()
        self.solver_se = optim.Adam(self.se_based.parameters(), lr=cfg['LR_SE'])
        self.scheduler_se = LS.MultiStepLR(self.solver_se,
milestones=cfg['TRAIN']['MILESTONES'], gamma=cfg['TRAIN']['GAMMA'])
        self.order = torch.randperm(cfg['CODE_CHNS']).cuda()
        _, self.reorder = torch.sort(self.order)

        def save_checkpoint(self, model_name):
        self.log.info('------------------------------Save Cpt------------------------------')

        ckp_path = osp.join(self.model_dir, model_name)

        self.log.info('Save checkpoint: %s' % ckp_path)
```

```
obj = {
'encoder': self.encoder.state_dict(),
'quantizer': self.quantizer.state_dict(),
'decoder': self.decoder.state_dict(),
'entropy': self.entropy.state_dict(),
'chn_imp': self.se_based.state_dict(),
'epoch': self.epoch
}
torch.save(obj, ckp_path)
self.log.info('Save the trained model successfully.')

def load_checkpoint(self, date, model_name):
self.log.info('-----------------------------Load Cpt-----------------------------')
ckp_path = osp.join(self.model_dir[:-12] + date, model_name)
try:
obj = torch.load(ckp_path, map_location=lambda storage, loc: storage.cuda())
self.log.info('Load checkpoint: %s' % ckp_path)
except IOError:
self.log.error('No checkpoint: %s!' % ckp_path)
return
self.encoder.load_state_dict(obj['encoder'])
self.quantizer.load_state_dict(obj['quantizer'])
self.decoder.load_state_dict(obj['decoder'])
self.entropy.load_state_dict(obj['entropy'])
self.se_based.load_state_dict(obj['chn_imp'])
self.epoch = obj['epoch']
self.log.info('The loaded model has been trained for %d epoch(s).' % self.epoch)


def update_lr(self):
self.scheduler_enc.step()
self.scheduler_qua.step()
self.scheduler_dec.step()
self.scheduler_etp.step()
self.scheduler_se.step()

def _train_init(self):
self.encoder.train()
for param in self.encoder.parameters():
param.requires_grad = True

self.quantizer.train()
for param in self.quantizer.parameters():
param.requires_grad = True

self.entropy.train()
for param in self.entropy.parameters():
param.requires_grad = True

self.decoder.train()
for param in self.decoder.parameters():
```

```
param.requires_grad = True

self.se_based.train()
for param in self.se_based.parameters():
param.requires_grad = True


def _eval_init(self):
self.encoder.eval()
for param in self.encoder.parameters():
param.requires_grad = False

self.quantizer.eval()
for param in self.quantizer.parameters():
param.requires_grad = False

self.entropy.eval()
for param in self.entropy.parameters():
param.requires_grad = False

self.decoder.eval()
for param in self.decoder.parameters():
param.requires_grad = False

self.se_based.train()
for param in self.se_based.parameters():
param.requires_grad = False

def _solver_init(self):
self.solver_qua.zero_grad()
self.solver_dec.zero_grad()
self.solver_etp.zero_grad()
self.solver_enc.zero_grad()
self.solver_se.zero_grad()

def _solver_update(self):
torch.nn.utils.clip_grad_value_(self.encoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.decoder.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.quantizer.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.entropy.parameters(), self.clip_value)
torch.nn.utils.clip_grad_value_(self.se_based.parameters(), self.clip_value)

self.solver_enc.step()
self.solver_qua.step()
self.solver_dec.step()
self.solver_etp.step()
self.solver_se.step()

def _order_feat(self, feat):
imp_value = self.se_based(feat)
mean_imp_value = torch.mean(imp_value, axis=0)
```

```
        _, self.order = torch.sort(mean_imp_value.squeeze())
        _, self.reorder = torch.sort(self.order)
        return feat[:, self.order, :, :]

        def _reorder_code(self, bc):
        return bc[:,self.reorder,:,:]
```

#main_train_eval.py

```
import argparse
import pprint
import numpy as np
np.set_printoptions(suppress=True)

from config import config
from config import update_config, create_logger
from trainer import *

def parse_args():
        parser = argparse.ArgumentParser(description='Train Deep Compression System: CVQN')

        parser.add_argument('--cfg',
                help='experiment configure file name',
                required=True,
                type=str)
        parser.add_argument('opts',
                help="Modify config options using the command-line",
                default=None,
                nargs=argparse.REMAINDER)

        args = parser.parse_args()
        update_config(config, args)

        return args

def main():
        args = parse_args()
        logger, model_dir, tb_log_dir = create_logger(config, args.cfg, 'train')
        logger.info(pprint.pformat(args))
        logger.info(config)

        if config['IMP_TYPE'] == 'predefine':
        trainer = PredefineTrainer(config, logger, model_dir, tb_log_dir, config['IMP_TYPE'])

        elif config['IMP_TYPE'] == 'RE':
        trainer = RETrainer(config, logger, model_dir, tb_log_dir, config['IMP_TYPE'])

        elif config['IMP_TYPE'] == 'SE':
        trainer = SETrainer(config, logger, model_dir, tb_log_dir, config['IMP_TYPE'])

        else:
```

```
        raise NotImplementedError("Trainer type error.")

        for epoch in range(config['TRAIN']['NUM_EPOCH']):
        trainer.train()
        trainer.eval()

        if config['IMP_TYPE'] == 'RE' and (epoch + 1) % 10 == 0:
        trainer.re_based_get_imp()

        if epoch == config['TRAIN']['NUM_EPOCH'] - 1:
        trainer.save_checkpoint('final.pth')

        trainer.update_lr()

        trainer.writer.close()
        trainer.metrics_print(config)


if __name__ == '__main__':
        main()
```

#dataset.py

```
# modified from
https://github.com/desimone/vision/blob/fb74c76d09bcc2594159613d5bdadd7d4697bb11/torchvisi
on/datasets/folder.py

import os
import os.path

import torch
from torchvision import transforms
import torch.utils.data as data
from PIL import Image
import pdb

IMG_EXTENSIONS = ['.jpg', '.JPG', '.jpeg', '.JPEG', '.png', '.PNG', '.ppm','.PPM', '.bmp',    '.BMP',]

def is_image_file(filename):
        return any(filename.endswith(extension) for extension in IMG_EXTENSIONS)

def default_loader(path):
        return Image.open(path).convert('RGB')

class ImageFolder(data.Dataset):
        """ ImageFolder can be used to load images where there are no labels."""

        def __init__(self, root, transform=None, loader=default_loader):
        images = []
        for filename in os.listdir(root):
        if is_image_file(filename):
```

```
            images.append('{}'.format(filename))

self.root = root
self.imgs = images
self.transform = transform
self.loader = loader

def __getitem__(self, index):
filename = self.imgs[index]
try:
img = self.loader(os.path.join(self.root, filename))
except:
return torch.zeros((3, 32, 32))

if self.transform is not None:
img = self.transform(img)
return img, filename

def __len__(self):
return len(self.imgs)
```